Patent Application
Docket No. 002.0160.01
NAI Docket No. 99.111.01

SYSTEM AND METHOD FOR IDENTIFYING A MACRO VIRUS FAMILY USING A MACRO VIRUS DEFINITIONS DATABASE

Field of the Invention

The present invention relates in general to macro virus identification and, in particular, to a system and a method for identifying a macro virus family using a macro virus definitions database.

Background of the Invention

Computer viruses, or simply "viruses," continue to plague unsuspecting users worldwide with malicious and often destructive results. Computer viruses propagate through infected files or objects and are often disguised as application programs or are embedded in library functions, macro scripts, electronic mail (email) attachments, applets, and even within hypertext links. Typically, a user unwittingly downloads and executes the infected file, thereby triggering the virus.

By definition, a computer virus is executable program code that is self-replicating and almost universally unsanctioned. More precisely, computer viruses include any form of self-replicating computer code which can be stored, disseminated, and directly or indirectly executed. The earliest computer viruses infected boot sectors and files. Over time, computer viruses evolved into numerous forms and types, including cavity, cluster, companion, direct action, encrypting, multipartite, mutating, polymorphic, overwriting, self-garbling, and

stealth viruses, such as described in "McAfee.com: Virus Glossary of Terms," http://www.mcafee.com/anti-virus/virus_glossary.asp?, Networks Associates Technology, Inc., Santa Clara, California (2000), the disclosure of which is incorporated by reference.

In particular, macro viruses have become increasingly popular, due in part to the ease with which these viruses can be written. Macro viruses are written in widely available macro programming languages and can be attached to document templates or electronic mail. These viruses can be easily triggered by merely opening the template or attachment, as graphically illustrated by the recent "Love Bug" and "Anna Kournikova" macro virus attacks in May 2000 and February 2001, respectively. The "Love Bug" virus was extremely devastating, saturating email systems worldwide and causing an estimated tens of millions of dollars worth of damage.

Today, there are over 53,000 known computer viruses and new viruses are being discovered daily. The process of identifying and cataloging new viruses is manual and labor intensive. Anti-virus detections companies employ full-time staffs of professionals whose only job is to analyze suspect files and objects for the presence of viruses. On average, training an anti-virus specialist can take six months or longer. These professionals are hard pressed to keep up with the constant challenge of discovering and devising solutions to new viruses.

In the prior art, few automated tools for identifying new viruses exist. On the front line, the processes employed by anti-virus experts to discover new viruses are *ad hoc* and primarily reactive, rather than proactive. Typically, suspect files or objects are sent to the virus detection centers by concerned users who have often already suffered some adverse side effect from a possible virus. In times past, virus detection centers had more time during which to identify and analyze viruses, and to implement patches and anti-viral measures that could be disseminated before widespread infection occurred. Today, however, viruses often travel by e-mail and other forms of electronic communication and can infect entire networks at an alarming rate. As a result, the present manual processes for

detecting new viruses are woefully slow and generally incapable of responding in a timely fashion.

Similarly, existing anti-virus software fails to provide an adequate solution to protecting and defeating new viruses. These types of software are designed to pattern scan and search out those viruses already positively identified by anti-virus software vendors. Invidious writers of computer viruses constantly strive to create new forms of viruses and easily evade existing anti-virus measures.

Therefore, there is a need for an approach to automatically identifying new forms of computer viruses and, in particular, macro computer viruses. Preferably, such an approach would be capable of identifying candidate virus families when presented with a suspect string or a particular virus family when presented with a suspect file or object. Moreover, such an approach would be capable of identifying a macro virus within a range of given search parameters.

Summary of the Invention

The present invention provides an automated system and method for maintaining and accessing a database of macro virus definitions. The database is organized by macro virus families, as characterized by replication method. In addition, the database stores string constants and source code text representative of and further characterizing macro families. A suspect string can be compared to the macro virus definitions maintained in the database to determine those macro virus families to which the string likely belongs. Similarly, a suspect file or object can be compared to the macro virus definitions in the database to determine the likely family to which the suspect file or object belongs. Thresholds specifying the percentage of common string constants and common text lines, as well as minimal length of string constants, can be specified.

An embodiment of the present invention is a system and a method for identifying a macro virus family using a macro virus definitions database. A macro virus definitions database is maintained and includes a set of indices and macro virus definition data files. Each index references one or more of the macro virus definition data files. Each macro virus definition data file defines macro virus attributes for known macro viruses. The sets of the indices and the macro

virus definition data files are organized according to macro virus families in each respective index and macro virus definition data file set. A suspect string is compared to the macro virus attributes defined in the one or more macro virus definition data files for each macro virus family in the macro virus definitions database. Each macro virus family to which the suspect string belongs is determined from the index for each macro virus definition data file at least partially containing the suspect string.

A further embodiment is a system and a method for identifying a macro virus family using a macro virus definitions database. A macro virus definitions database is maintained and includes a set of indices and associated macro virus definition data files. One or more of the macro virus definition data files are referenced by the associated index. Each macro virus definition data file defines macro virus attributes for known macro viruses. The sets of the indices and the macro virus definition data files are organized according to macro virus families. One or more strings stored in a suspect file are compared to the macro virus attributes defined in the one or more macro virus definition data files for each macro virus family in the macro virus definitions database. The macro virus family to which the suspect file belongs is determined from the indices for each of the macro virus definition data files at least partially containing the suspect file.

Still other embodiments of the present invention will become readily apparent to those skilled in the art from the following detailed description, wherein is described embodiments of the invention by way of illustrating the best mode contemplated for carrying out the invention. As will be realized, the invention is capable of other and different embodiments and its several details are capable of modifications in various obvious respects, all without departing from the spirit and the scope of the present invention. Accordingly, the drawings and detailed description are to be regarded as illustrative in nature and not as restrictive.

Brief Description of the Drawings

FIGURE 1 is a functional block diagram of a distributed computing environment, including a system for identifying a macro virus family using a macro virus definitions database, in accordance with the present invention.

FIGURE 2 is a block diagram of the system for identifying a macro virus family of FIGURE 1.

FIGURE 3 is a block diagram showing the software modules implemented in the system of FIGURE 1.

FIGURE 4 is a data structure diagram showing the cataloging of macro virus definitions.

FIGURE 5 is a data structure diagram showing a parse tree header.

FIGURE 6 is a data structure diagram showing a strings block.

FIGURE 7 is a data structure diagram showing, by way of example, a parse tree constructed using the data structures of FIGURES 5 and 6.

FIGURE 8 is a flow diagram showing a method for identifying a macro virus family using a macro virus definitions database in accordance with the present invention.

FIGURES 9A-9C are flow diagrams showing the routine for finding a macro virus family for use in the method of FIGURE 8.

FIGURES 10A-10B are flow diagrams showing the routine for finding a string for use in the method of FIGURE 8.

FIGURES 11A-11C are flow diagrams showing the routine for updating the virus definitions database for use in the method of FIGURE 8.

FIGURES 12A-12D are flow diagrams showing the routine for checking the virus definitions database for use in the method of FIGURE 8.

FIGURES 13A-13B are flow diagrams showing the routine for listing the macro virus definitions.

Detailed Description

FIGURE 1 is a functional block diagram showing a distributed computing environment 10, including a system for identifying a macro virus family 16, using a macro virus definitions database, in accordance with the present invention. The

networked computing environment 10 includes one or more servers 12 interconnected to one or more clients 13 over an internetwork 11, such as the Internet. Each server 12 provides client services, such as information retrieval and file serving. Alternatively, the clients could be interconnected with the server 12 using a direct connection, over a dial-up connection, via an intranetwork 14, by way of a gateway 15, or by a combination of the forgoing or with various other network configurations and topologies, as would be recognized by one skilled in the art.

A client 13, or alternatively a server 12, implements a macro virus checker (MVC) 16 for identifying macro virus attributes using a macro virus definitions database, as further described below with reference to FIGURE 2. During operation, a user can submit a suspect string to the macro virus checker 16 to identify candidate virus families to which the suspect string may belong. Alternatively, the user can submit a file or object to the macro virus checker 16 to identify a candidate virus family to which the suspect file or object belongs.

The individual computer systems, including the servers 12 and clients 13, are general purpose, programmed digital computing devices consisting of a central processing unit (CPU), random access memory (RAM), non-volatile secondary storage, such as a hard drive or CD ROM drive, network interfaces, and peripheral devices, including user interfacing means, such as a keyboard and display. Program code, including software programs, and data are loaded into the RAM for execution and processing by the CPU and results are generated for display, output, transmittal, or storage.

FIGURE 2 is a block diagram showing the system for identifying a macro virus family of FIGURE 1. By way of example, the macro virus checker 16 executes on a client 13 coupled to a secondary storage device 17. The system is preferably implemented in software as a macro virus checker 16 operating on the client 13, or on the server 12 (shown in FIGURE 1) or any similar general purpose programmed digital computing device. The storage device 17 includes a file system 18 within which files and related objects are persistently stored. In addition, the client 13 interfaces to other computing devices and resources via an

intranetwork 14, an internetwork 11 (shown in FIGURE 1), or other type of network or communications interface.

FIGURE 3 is a block diagram showing the software modules implementing the macro virus checker 16 of the system of FIGURE 1. Each module is a computer program, procedure or module written as source code in a conventional programming language, such as the C++ programming language, and is presented for execution by the CPU as object or byte code, as is known in the art. The various implementations of the source code and object and byte codes can be held on a computer-readable storage medium or embodied on a transmission medium in a carrier wave. The macro virus checker 16 operates in accordance with a sequence of process steps, as further described below beginning with reference to FIGURE 8. The Appendix includes a source code listing for a computer program in the C++ programming language implementing the macro virus checker 16.

The macro virus checker 16 consists of six intercooperating modules: parser 20, family finder 21, string finder 22, updater 23, checker 24, and lister 25. Operationally, the macro virus checker 16 receives as an input either a suspect string 26 or a suspect file 27 or object (hereinafter simply "suspect file") for comparison to the database of macro virus definitions 28. The suspect string 26 or suspect file 27 is parsed by the parser 20 to identify individual tokens. In the described embodiment, the parser 20 removes comments and extraneous information from the suspect string 26 and suspect file 27. The parser 20 processes the suspect string 26 and suspect file 27 on a line-by-line basis and generates a hierarchical parse tree, as is known in the art.

During analysis, a suspect string 26 or suspect file 27 (shown in FIGURE 3) is parsed into individual tokens stored in a parse tree. As further described below with reference to FIGURE 7, parse tree stores individual string constants and source code text as two linked lists rooted using a parse information header.

Once parsed, a number of operations can be performed on the parse tree. First, the macro virus family to which the suspect file 27 belongs can be identified using the family finder 21, as further described below with reference to FIGURES

9A-9C. Similarly, the candidate macro virus families to which the suspect string 26 belongs can be identified by the string finder 22, as further described below with reference to FIGURES 9A-9C. The macro virus definitions database 28 can be updated using the updater 23, as further described below with reference to FIGURES 10A-10B. Likewise, the macro virus definitions database 28 can be checked for cross-references using the checker 24, as further described below with reference to FIGURES 12A-12D. Finally, the file names of the macro virus definition families can be listed using the lister 25, as further described below with reference to FIGURES 13A-13B.

The macro virus definitions database 28 is hierarchically organized into macro virus families based on the type of application to which the macro applies. By way of example, the macro virus definitions database 28 can include a root directory 29, below which word processor 30, spreadsheet 31, presentation 32, and generic 33 subdirectories can contain individual indices and macro virus definition data (*.dat*) files, as further described below with reference to FIGURE 4. The results of the operations performed by the macro virus checker 16 on the suspect string 26 or suspect file 27 are output in a report 35 and details of the analysis are provided in a log file 34.

FIGURE 4 is a data structure diagram 40 showing the indexing of a macro virus definitions family. An index maintained in index files, *route.idx* 41 stores pointers to locations in individual *.dat* files *000000001.dat* 42, *000000002.dat* 43 and *000000002.dat* 44 files. Each of the *.dat* files 42-44 store information describing a macro virus family, as characterized by the replication method used by the virus. In the described embodiment, the replication methods include types "organizer," "macro copy," "import," "replace line," "insert lines," "add from string," and "add from file."

In addition, each *.dat* file contains any string constants and lines of source code text, without comments, common to all replicants of the macro virus. The macro virus definition is assigned a name to aid in the understanding by the user. Macro viruses are further described in M. Ludwig, "The Giant Black Book of

Computer Viruses," Ch. 14, American Eagle Pubs, Inc., Show Low, AZ (2nd ed. 1998), the disclosure of which is incorporated by reference.

FIGURE 5 is a data structure diagram showing the structure of the header 50 *TparseInfo* for storing parse information. The header includes a count of the number of files *FilesNUM* from which the suspect file 27 originates, pointers to the string constants *Strings* and source code text *Lines*, an index to the first string for the string constants *TopString*, an index to the first string for the source code text *TopLine*, and a count of the number of strings *StringsNum* and source code text *LinesNum*. Finally, the parse information header includes a byte flag *ReplFlags* storing an indication of the type of replication method used.

FIGURE 6 is a data structure diagram showing the structure of each node *TStrings* 60 in which each of the sets of parsed tokens for the string constants and source code text are stored. The actual token is stored as a character string *String* along with the type and use of the string. A pointer *Next* points to the next node in the linked list.

FIGURE 7 is a data structure diagram showing, by way of example, a parse tree 70 for a suspect file 27 (shown in FIGURE 3). The parse information header *TParseInfo* 71 points to the first node 73a-d, 75a-e in each of the respective linked lists for the main constants *Strings* 72 and source code text *Lines* 74. Each of the individual nodes in the strings linked list 72 and lines linked list 74 point to the next node in each list. The linked lists wrap back around such that each list forms a continuous chain. The first string (for string constants) or index (for source code text) in each chain is respectively identified by a counter *TopString* or *TopLine*, as further described above with reference to FIGURE 5.

FIGURE 8 is a flow diagram showing a method 80 for identifying macro virus attributes using macro virus definitions database 28 (shown in FIGURE 3) in accordance with the present invention. The method provides an environment in which the macro virus definitions database 28 can be maintained and accessed to determine macro virus attributes and family membership for a suspect string 26 or a suspect file 27.

The method 80 begins with the initialization of a working environment. First, the storage file, that is, the directory containing the macro family description datafile, is opened (block 81). Next, the log file 34 (shown in FIGURE 3) is set (block 82) and the initialization file is opened (block 83). Any parameters specified by the user are set, in addition to any default parameters (block 84). Processing then begins.

The macro virus checker 16 performs several operations based on a user or automatically specified selection (blocks 85-92) as follows. First, a full report can be generated (block 86) to present the macro virus definition family stored in the macro virus definitions database 28. A macro virus family can be found for a suspect file 27 (block 87), as further described below with reference to FIGURES 9A-9C. A set of macro virus families containing a given string can be found (block 88), as further described below with reference to FIGURES 10A-10B. The macro virus definitions database 28 can be updated (block 89), as further described below with reference to FIGURES 11A-11C. Similarly, the macro virus definitions database 28 can be checked for cross-references (block 90), as further described below with reference to FIGURES 12A-12D. Finally, the macro virus definition families can be listed (block 91), as further described below with reference to FIGURES 13A-13B. The method terminates upon the completion of the various operations.

FIGURES 9A-9C are flow diagrams showing the routine for finding a macro virus family 100 for use in the method of FIGURE 8. The purpose of this routine is to identify, if possible, the macro virus family to which a suspect file 27 (shown in FIGURE 3) belongs. The user can specify a given confidence level representing a percentage for string constants and the matches for the replication method used. The routine will determine the closest matching macro virus family within the given search parameters.

First, the suspect file 27 is parsed (block 101) and the log file is set (block 102). A found array is initialized (block 103) within which matching common string constants and common text lines are stored. A search entry is set to the first entry in the parse tree (block 104). Each entry in the parse tree is iteratively

processed (blocks 105-125), as follows. First, an index file 41 (shown in FIGURE 4) is opened (block 106) and a list of strings stored therein is obtained (block 107). The list of strings is indexed by a current index pointer set to the first string in the chain (block 108). Each of the strings is then iteratively processed (blocks 109-114), as follows. First, a token from the parse tree is compared to the string for matching or partially matching a string constant (block 110). If the token matches (block 111), a same string counter is incremented (block 112). The current index is set to the next index in the chain (block 113) and processing of the current list of strings continues until the string is complete.

Next, if the detection level for source code text is greater than zero (block 115), the token is also compared to any stored source code text (blocks 116-122). Otherwise, no source code text comparisons are performed. Thus, assuming source code text is also being searched, the current index is set to the first index in the chain (block 116) and each of the nodes of source code text in the linked list are iteratively processed (blocks 117-122), as follows. A token from the parse tree is compared to the source code text (block 118). If the token matches (block 119), a same text counter is incremented (block 120). The current index is set to the next index in the chain (block 121) and iterative processing continues (block 117) until the list of text is complete.

Next, the results of the searches for matching string constants and, if performed, source code text, are saved (block 123) and the search entry is set to the next entry in the parse tree (block 124). Each of the parse tree nodes is processed (block 125) until the parse tree is complete. Finally, a report is output (block 126) indicating the results of the search, after which the routine returns.

FIGURES 10A-10B are flow diagrams showing the routine for finding a string 130 for use in the method of FIGURE 8. The purpose of this routine is to find those macro virus definition families in which a suspect string 26 (shown in FIGURE 3) can be found. This routine functions as an adjunct to the routine for finding a macro virus definition family 100 (shown in FIGURES 9A-9C), as a suspect file 27 consists of one or more suspect strings 26 and the results of the

more extensive searching performed by the find family routine 100 can narrow down the field to a single macro virus definition family.

As before, the log file 34 (shown in FIGURE 3) is set (block 131) and the search entry is set to the first entry in the parse tree (block 132). The parse tree is iteratively processed (block 133-142), as follows. First, an index file 41 (shown in FIGURE 4) is opened (block 134) and a found flag is set to the first replication byte flag *ReplFlags* (shown in FIGURE 5) (block 135). Recall that the byte flag *replFlag* indicates the replication method used by the macro virus family. Each byte flag *ReplFlags* is iteratively processed (136-140), as follows.

First, the *.dat* file 42 (shown in FIGURE 4) is opened (block 137) and each line containing the source code text identified by the current token is found (block 138). The byte flag is set to the next byte flag *ReplFlags* (block 139) and iterative processing continues until all of the byte flags *ReplFlags* are complete (block 136). The search entry is then set to the next entry in the parse tree (block 141) and iterative processing continues through the parse tree until the parse tree is complete (block 133). Finally, a report is output (block 143), after which the routine returns.

FIGURES 11A-11C are flow diagrams of the routine for updating the macro virus definitions database 28 (shown in FIGURE 3) for use in the method of FIGURE 8. The purpose of this routine is to update and index any new macro virus definitions into the macro virus definitions database 28.

First, the log file 34 (shown in FIGURE 3) is set (block 151). Each entry in the macro virus definitions database 28 is iteratively processed as follows. First, the first entry in the database 28 is obtained (block 152) and iteratively processed (blocks 153-174) as follows. The index file 41 (shown in FIGURE 4) is reset (block 154) and the first item to scan is found (block 155) and iteratively processed (blocks 156-171) as follows. The parser 20 (shown in FIGURE 3) is initialized (block 157) and the scan item, that is, macro virus file, is parsed (block 158) to generate a parse tree 70 (shown in FIGURE 7). The item header that is storing the parse information 50 (shown in FIGURE 5) is stored (block 159). Each of the chains of nodes storing string constants and source code text are

processed (blocks 160-164 and 165-169, respectively). The string constants are processed first by setting the current index to the first index in the chain of string constants 72 (shown in FIGURE 7) (block 160). Each of the indexes is iteratively processed (block 161-169) as follows. Each string constant *Strings* (shown in FIGURE 5) is stored using the current index as an index into the *Strings* array (block 162). The current index is then set to the next index in the chain of strings 72 (block 163). Next, each of the source code text segments is processed by setting the current index to the first index in the chain of source code text segments 74 (shown in FIGURE 7) (block 165). The source code text segments 74 are iteratively processed (blocks 166-169), as follows. Each source code text segment is stored in the *Lines* array indexed by the current index (block 167). The current index is then set to the next index in the chain of source code text segments 74 (block 168).

After all of the string constants and source code text segments are processed (blocks 160-164 and 165-169, respectively), the next scan item, that is, macro virus file, is obtained (block 170) and iteratively processed (blocks 156-171), as follows. Next, the index file 41 (shown in FIGURE 4) is closed (block 172) and the next entry in the database 28 is obtained (block 173). Processing of database entries continues (blocks 153-174) until the database 28 is complete, after which the routine returns.

FIGURES 12A-12D are flow diagrams showing the routine for checking the macro virus definitions database 28 (shown in FIGURE 3) for use in the method of FIGURE 8. The purpose of this routine is to check for cross references in the macro virus definition database 28.

Each of the entries in the database 28 are iteratively processed (blocks 182-217) after first obtaining the first entry in the database 28 (block 181). The index file 41 (shown in FIGURE 4) for the current database entry is opened (block 183). Each of the scan items, that is, macro virus definitions, is iteratively processed (blocks 185-214) after first selecting the first scan item (block 184). Similarly, each file object, that is, macro virus file, is iteratively processed (blocks 187-212) after first selecting a first file object (block 186).

During the processing of each file object, the parser 20 (shown in FIGURE 3) is initialized (block 188) and the file object is parsed (189) to generate a parse tree 70 (shown in FIGURE 7).

Next, each of the macro virus families, as characterized by their respective methods of replication, is processed as follows. The types of replication methods are indicated in the byte flag *ReplFlags* (shown in FIGURE 5). Each of the macro virus definition families is iteratively processed (blocks 191--21) after first selecting the first byte flag *ReplFlags* (block 190). If the current file object is in the same macro virus replication family (block 192), the family is skipped. Otherwise, the *.dat* file 42 (shown in FIGURE 4) is processed as follows.

For each *.dat* file 42, the string constants and source code text segments are processed (blocks 194-200 and 202-208, respectively). First, the current *.dat* file is opened (block 193). Next, the current index is set to the first index in the chain of string constants (block 194) and iterative processing (block 195) begins. The strong is compared to the string constants for the current macro virus definition (block 196), and if the string matches (block 197), the same string counter is incremented (block 198). The current index is set to the next index in the chain of string constants 72 (block 199) and iterative processing continues (block 195) until the chain of string constants is complete. Next, if the detection level for text is greater than zero (block 201), source code text segments are processed as follows. First, the current index is set to the first index in the chain of source code text segments 74 (shown in FIGURE 7). Iterative processing then begins (block 203). The string is compared to the source code text segments stored in the current macro virus definition (block 204), and if a match is found (block 205), the same text counter is incremented (block 206). The index is set to the next index in the chain of source code text 74 (block 207). Iterative processing continues (block 204) until the chain of source code segments 74 is complete.

The string constants and source code text having been processed, the next macro virus family is selected by setting the found flag to the next byte flag

ReplFlags (block 209) and the macro virus definition families are iteratively processed (block 191) until the families are complete.

Similarly, the next file object is selected (block 211) and the file objects are iteratively processed (block 187) until all the file objects are complete. Next, the next scan item, that is, *.dat* file 43 (shown in FIGURE 4) (block 213) for each of the scan items is iteratively processed (block 185) until the scan items are complete. Finally, the index file 41 (shown in FIGURE 4) is closed (block 215) and the next entry in the macro virus definitions database 28 (shown n FIGURE 3) is selected (block 216). Each of the macro virus definition database 28 entries is iteratively processed (block 182) until the database entries are complete, after which the routine returns.

FIGURES 13A-13B are flow diagrams showing the routine for listing the macro virus definition families in the database 28 (shown in FIGURE 3) for use in the method of Figure 8. The purpose of this routine is to iteratively list the macro virus families.

Each of the entries in the database 28 is iteratively processed (blocks 222-235) by first selecting the first entry in the database 28 (block 221). The index file 41 (shown in FIGURE 4) is open (block 223). A found flag is set to the first byte flags *replFlag* (shown in FIGURE 5) (block 224) to indicate the current macro virus definition family. Recall that the macro virus definition families are identified by replication method. Iterative processing begins (block 225) by walking through the parse information headers 50 (shown in FIGURE 5) (blocks 226-230), as follows. First, a head pointer is set to the current headers (block 227) and, if the header has not been printed (block 228), the index offset, header level, name, replication flags, next sibling, cluster and *.dat* offset are printed (block 229). Upon completion of the printing of each of the headers (blocks 226-230), the next macro virus family is selected by setting the found flag to the next byte flags *replFlag* (block 230). Iterative processing continues with the next macro family (block 225) after which the index file 41 is closed (block 233) and the next entry in the database 28 is selected (block 234). Iterative processing of

database entries continues (block 222) until all of the database entries are complete, after which the routine returns.

While the invention has been particularly shown and described as referenced to the embodiments thereof, those skilled in the art will understand that the foregoing and other changes in form and detail may be made therein without departing from the spirit and scope of the invention.

```
/**************************************************
* MacroFam.cpp
**************************************************
* Author: Viatcheslav Peternev (Network Associates,Inc)
*
* Description:- Main module of programm for
*               determination of macrofamily
*
**************************************************/
// 1.01 - fixed starutup name for NT
// 1.02 - added type "generic vba4fixed starutup name for NT

#include <direct.h>

#include "t_app.h"
#include "t_fobj.h"

#include "MFamDefs.h"

//--- Definitions
#define MAX_TEXTBUF_LEN       4096
#define MAX_BINBUF_LEN 4096
#define MAX_PATHNAME_LEN      512
#define MAX_STRING_LEN        255
#define MAX_DECOMPR_LEN       4096


//--- Global vars
TApp    g_App;

char    *g_Header[]=
{
    "MacroFam  ver.1.02 -  determination of macro family  by V.Peternev",
    "(C) 1999,2000 Network Associates,Inc",
       NULL
};
char    *g_HelpHead[]=
{
    "",
    "  Usage:   macrofam filename [/options] ",
    "",
    "    filename - input file",
       NULL
};
char    *g_HelpFoot[]=
{
    "  Example:",
    "    macrofam vir.doc  - check file for best matched family",
    "        (subdirectory MFAMBASE is in the directory with macrofam.exe)",
    "    macrofam $string  - find families with given string",
       NULL
```

```
};
enum{PARAM_NAME,              PARAM_BASE,                   PARAM_LOG,            PARAM_INI,                    PARAM_SRC,
        PARAM_NUMBEST,              PARAM_LEVELREPL,      PARAM_LEVELSTRING,    PARAM_LEVELTEXT,PARAM_STRINGMINLEN,
        PARAM_UPDATE,         PARAM_CHECK,          PARAM_FROM,           PARAM_GURU,           PARAM_LIST};

TAppParam g_Params[] =
{
        {PARAM_NAME,          0, NULL,       NULL,                  ""},
        {PARAM_BASE,          1, "BASE",     "mfambase", "directory with macrofam data (default = startup)"},
        {PARAM_LOG,                 1, "REP",      "macrofam.rep", "report filename (default = macrofam.rep)"},
        {PARAM_NUMBEST,             1, "NUM",      "1", "number of best matches for report (default=1)"},
        {PARAM_INI,                 2, "INI",      "macrofam.ini", "INI filename"},
        {PARAM_GURU,          2, "GURU",     NULL, "full report"},
        {PARAM_LEVELREPL,     2, "LR",       "0", "match replication method (0/1 ,def=0)"},
        {PARAM_LEVELSTRING,   2, "LS",       "50", "percent of common string constants (def=50)"},
        {PARAM_LEVELTEXT,     2, "LT",       "20", "percent of common text lines (def=20)"},
        {PARAM_STRINGMINLEN,2, "SLEN",       "0",  "minimal length of string constants (def=0)"},
        //{PARAM_SRC,                 1, "SRC",      "src", "directory for sources of found familes"},
        {PARAM_UPDATE,        2, "UPDATE",NULL, "rebuild database by scanning the collection"},
        {PARAM_CHECK,         2, "CHECK",NULL, "check for cross references in the collection"},
        {PARAM_LIST,          2, "LIST", "macrofam.lst", "families list filename (def=macrofam.lst)"}
};
BYTE    g_NumParam = sizeof(g_Params) / sizeof(TAppParam);

// functions
int ProcReports(char *InName, char *OutName, char *DrvName);
int ProcOneReport(TTextFile &InFile, TTextFile &OutFile, TTextFile &DrvFile);
int     LoadTextPart(char *TextBuf, int MaxTextLen,  char *Line);
int FormBinPart(char *TextBuf, BYTE *BinBuf, int MaxBinLen);
void FormOutText(TTextFile &OutFile, BYTE *BinBuf, int BinLen);

/*************************************************
* main() -Main function for module
*************************************************
* [OUT] =0 - OK
*       <0 - error code
*
*************************************************/
int main(
    int argc,       // arguments counter
    char *argv[]    // arguments array
    )
{
        g_App.SetParam(g_Params, g_NumParam);
        g_App.SetHelpDescr(g_HelpHead, g_HelpFoot);
        g_App.SetHeader(g_Header);

        if (g_App.ProcParam(argc,argv))
        {
                TMacroFamDefs Macro families;
                BOOL    bOpen;
```

```
//printf("\nStartName=%s",g_App.m_StartFileName);

if ( g_App.WasParam(PARAM_BASE) )
        bOpen = Macro families.OpenStorage( g_App.GetStrParamValue(PARAM_BASE) );
else
{       // try default
        sprintf( g_App.m_LineBuf, "%s\\%s", g_App.GetStartupDir(), g_App.GetStrParamValue(PARAM_BASE));
        bOpen = Macro families.OpenStorage( g_App.m_LineBuf);
}
if (!bOpen)
{
        printf("\n%s\n", Macro families.ErrMess( Macro families.ErrCode() ) );
        return -1;
}
if (g_App.WasParam(PARAM_LOG))
        Macro families.SetLogFile( g_App.GetStrParamValue(PARAM_LOG) );

FILE *IniFile = g_App.OpenIniFile( g_App.GetStrParamValue(PARAM_INI), TRUE);

// Set defaults param & from ini-file
Macro families.SetParams( IniFile );

int iParam;

if (g_App.WasParam(PARAM_NUMBEST))
{
        sscanf(g_App.GetStrParamValue(PARAM_NUMBEST), "%d", &iParam);
        Macro families.m_NumBest = iParam;
}
if (g_App.WasParam(PARAM_GURU))
        Macro families.m_bGuru = TRUE;

if (g_App.WasParam(PARAM_LEVELREPL))
{
        sscanf(g_App.GetStrParamValue(PARAM_LEVELREPL), "%d", &iParam);
        Macro families.m_DetectLevelRepl = iParam;
}
if (g_App.WasParam(PARAM_LEVELSTRING))
{
        sscanf(g_App.GetStrParamValue(PARAM_LEVELSTRING), "%d", &iParam);
        Macro families.m_DetectLevelString = iParam;
}
if (g_App.WasParam(PARAM_LEVELTEXT))
{
        sscanf(g_App.GetStrParamValue(PARAM_LEVELTEXT), "%d", &iParam);
        Macro families.m_DetectLevelText = iParam;
}
if (g_App.WasParam(PARAM_STRINGMINLEN))
{
        sscanf(g_App.GetStrParamValue(PARAM_STRINGMINLEN), "%d", &iParam);
```

```
                Macro families.m_StringMinLen = iParam;
        }

        if (g_App.WasParam(PARAM_UPDATE))
        {       // scan for new families
                Macro families.Update();
        }
        else if (g_App.WasParam(PARAM_CHECK))
        {       // scan for new families
                Macro families.Check();
        }
        else if (g_App.WasParam(PARAM_LIST))
        {       // scan for new families
                Macro families.List( g_App.GetStrParamValue( PARAM_LIST));
        }
        else
        {
                char *FileName = g_App.GetStrParamByPos( 0 );
                if ( FileName )
                {
                        if ( *FileName != '$')
                        {       // determ. family
                                if (!Macro families.FindFamily( FileName ))
                                {
                                        printf("\n%s\n", Macro families.ErrMess( Macro families.ErrCode() ) );
                                        g_App.m_RetCode = -1;
                                }
                        }
                        else
                        {       // find string
                                if (!Macro families.FindString( FileName+1))
                                {
                                        printf("\n%s\n", Macro families.ErrMess( Macro families.ErrCode() ) );
                                        g_App.m_RetCode = -1;
                                }
                        }
                }
        }
    }
    else
            g_App.m_RetCode = -1;

return g_App.m_RetCode;
}
```

```
// mfamdefs.cpp - main class for macrofam
// Author - Viatcheslav Peternev (Network Associates,Inc)
//=========================================================
#include "MFamDefs.h"

enum {IniSectDetectLevel, IniSectStrings, IniSectText};
enum {IniKeyReplLevel, IniKeyStringLevel, IniKeyTextLevel, IniKeyStringMinLen };

typedef struct tagFindInfo{
        int                             Rate;
        int                             Next;
        char                    String[255];
} TFindInfo;

// ini sections
TIniSectDef     s_IniSections[] =
{
        { IniSectDetectLevel, "DetectLevel" },
        { IniSectStrings, "Strings" },
        { IniSectText, "Text" }
};
int     s_IniSectNum = sizeof (s_IniSections) / sizeof (TIniSectDef);

// ini keys
TIniKeyDef      s_IniKeys[] =
{
        { IniSectDetectLevel,  IniKeyReplLevel,       "Replication" },
        { IniSectDetectLevel,  IniKeyStringLevel,     "Strings" },
        { IniSectDetectLevel,  IniKeyTextLevel,       "Text" },
        { IniSectStrings,              IniKeyStringMinLen, "MinLength" }
};
int     s_IniKeyNum = sizeof (s_IniKeys) / sizeof (TIniKeyDef);
//=========================================================
TMacroFamDefs::TMacroFamDefs()
{
        m_Storage       = new TMacroFamStorage;
        m_Parser        = new TMacroFamParser;
        m_LogFile       = NULL;
        strcpy(m_ErrMessage, "Undefined error");
}
//=========================================================
// destructor
TMacroFamDefs::~TMacroFamDefs()
{
        if (m_Storage)
                delete m_Storage;
        if (m_Parser)
                delete m_Parser;
        if (m_LogFile)
                fclose(m_LogFile);
}
```

```
//=========================================================
bool TMacroFamDefs::OpenStorage( char *Name )
{
        bool bRet = m_Storage->Open(Name);

        if (!bRet)
                strcpy(m_ErrMessage, m_Storage->ErrMess( m_Storage->ErrCode()));

        return bRet;
}
//=========================================================
bool TMacroFamDefs::SetLogFile( char *Name )
{
        m_LogFile = fopen(Name, "w");

        return m_LogFile != NULL;
}
//=========================================================
void TMacroFamDefs::SetParams( FILE *inifile )
{
        // first set default
        m_DetectLevelRepl      = 1;
        m_DetectLevelString = 50;
        m_DetectLevelText      = 20;
        m_StringMinLen         = 0;
        m_NumBest                       = 1;
        m_bGuru                         = FALSE;

        if (inifile)
        {       // take from ini-file
                TIniFile *ini = new TIniFile;
                int SectCode, KeyCode;
                char *Val;

                ini->Assign( inifile );
                ini->SetSections( s_IniSections, s_IniSectNum);
                ini->SetKeys( s_IniKeys, s_IniKeyNum);

                while(Val = ini->GetNextVal(SectCode, KeyCode))
                {
                        switch (KeyCode)
                        {
                        case IniKeyReplLevel:
                                sscanf(Val, "%d", &m_DetectLevelRepl);
                                break;
                        case IniKeyStringLevel:
                                sscanf(Val, "%d", &m_DetectLevelString);
                                break;
                        case IniKeyTextLevel:
                                sscanf(Val, "%d", &m_DetectLevelText);
                                break;
```

```
                        case IniKeyStringMinLen:
                                sscanf(Val, "%d", &m_StringMinLen);
                                break;
                        default:
                                break;
                        }
                }
                delete ini;
        }
}
//=========================================================
bool TMacroFamDefs::FindFamily( char *FileName )
{
        bool bRet = true,
                bSearchEntry;
        TParseInfo *pInfo;
        int     CurrIndex, InsertIndex, ListIndex, iBeg, iEnd, i,
                FindWorstRate, NumBest, MinRate, MaxRate, MaxIndex,
                //PrevIndex, FindTopIndex,
                StringLevel,StringCount,
                TextLevel,  TextCount,
                CurrLevel,      iSame,          iSameText,
                Count;

        if (_access( FileName, 0 ) != 0)
        {
                sprintf(m_ErrMessage, "File %s not found!", FileName);
                return FALSE;
        }

        m_Parser->Init();
        if (!m_Parser->ParseFile( FileName ))
        {
                sprintf(m_ErrMessage, "Not found VBA modules in %s", FileName);
                return FALSE;
        }

        pInfo = m_Parser->GetParseInfo();

        m_Storage->SetLogFile( m_LogFile );

        if (m_DetectLevelRepl == 0)
                pInfo->ReplFlags = 0;           // reset all methods

        // init found array
        m_NumBest = __min(m_NumBest, 64);

        TFindInfo *BestFinds = new TFindInfo[ m_NumBest ];
        BOOL    bResult = FALSE;

        Count = 0;
```

```
for (i=0; i < m_NumBest; i++)
{
        BestFinds[ i].Next = 0;
        BestFinds[ i].Rate = 0;
}
FindWorstRate = 0;
NumBest = 0;


bSearchEntry = m_Storage->SetEntry( m_Parser->Type() );
if (!bSearchEntry)
{
        sprintf(m_ErrMessage, "Not defined type %s", m_Parser->Type() );
        return FALSE;
}
while (bSearchEntry)
{
        if (!m_Storage->OpenIndexFile() )
                return FALSE;

        if (m_LogFile && m_bGuru)
        {       // list of strings
                ListIndex = 0;
                CurrIndex = pInfo->TopString;
                fprintf(m_LogFile, "\nStrings in file %s", FileName);
                while ( CurrIndex >= 0)
                {
                        fprintf(m_LogFile, "\n%02d: \"%s\"", ListIndex, pInfo->Strings[ CurrIndex ].String);
                        ListIndex++;
                        CurrIndex = pInfo->Strings[ CurrIndex ].Next;        // next index in chain
                }
        }

        BOOL bFound = m_Storage->GetFirstByFlags( pInfo->ReplFlags );
        while ( bFound)
        {
                //printf("\n%d: %s", Count, m_Storage->FamilyName());

                //if (Count >=345)
                //      Count += 0;

                // compare string
                iSame = 0;
                CurrIndex = pInfo->TopString;
                ListIndex = 0;

                if (m_bGuru)
                        memset(m_FoundMap, 0, sizeof(m_FoundMap));

                m_Storage->OpenDatFile();
```

```
StringCount = 0;
while ( CurrIndex >= 0)
{
        if (strlen(pInfo->Strings[ CurrIndex ].String) >= (size_t)m_StringMinLen)
        {
                if ( m_Storage->FindString( pInfo->Strings[ CurrIndex ].String ) )
                {
                        iSame++;
                        if (m_bGuru)
                        {
                                if (ListIndex < sizeof(m_FoundMap))
                                        m_FoundMap[ListIndex] = 1;
                        }
                }
                StringCount++;
        }
        CurrIndex = pInfo->Strings[ CurrIndex ].Next;        // next index in chain
        ListIndex++;
}
StringLevel =  StringCount != 0 ? (iSame * 100)/ StringCount : 0;

// compare text
iSameText = 0;
TextCount = 0;
if ( m_DetectLevelText > 0)
{
        // read Lines counr
        m_Storage->ReadLineNum();

        CurrIndex = pInfo->TopLine;
        while ( CurrIndex >= 0)
        {
                if ( m_Storage->FindLine( pInfo->Lines[ CurrIndex ].String ) )
                {
                        iSameText++;
                }
                TextCount++;
                CurrIndex = pInfo->Lines[ CurrIndex ].Next;  // next index in chain
        }
}
TextLevel = TextCount != 0 ? (iSameText * 100)/ TextCount : 0;

if (!m_bGuru || (StringLevel >= m_DetectLevelString && TextLevel >= m_DetectLevelText))
{
        CurrLevel = (TextLevel + StringLevel) / 2;

        if (m_LogFile && m_bGuru)
        {
                fprintf(m_LogFile, "\n\n Found: %-30s Strings:%d/%d  Text:%d/%d",
                        m_Storage->FamilyName(),
```

```
                iSame, StringCount, iSameText, TextCount);
        fprintf(m_LogFile, "\n\tStrings: ");
        iBeg = iEnd = -2;
        for (i=0; i < sizeof(m_FoundMap); i++)
        {
                if ( m_FoundMap[i] == 1)
                {
                        if (i==(iEnd+1))
                                iEnd++;
                        else
                        {
                                // print prev.diap
                                if (iBeg >= 0)
                                {
                                        if (iEnd==iBeg)
                                                fprintf(m_LogFile, " %d", iBeg);
                                        else
                                                fprintf(m_LogFile, " %d-%d", iBeg, iEnd);
                                }
                                // begin new diap
                                iBeg = iEnd = i;
                        }
                }
        }
        // print last.diap
        if (iBeg >= 0)
        {
                if (iEnd==iBeg)
                        fprintf(m_LogFile, " %d", iBeg);
                else
                        fprintf(m_LogFile, " %d-%d", iBeg, iEnd);
        }
}
// save best results
if (m_NumBest == 1)
{
        if (CurrLevel >= BestFinds[0].Rate)
        {
                if (CurrLevel > 0)
                        printf("\n%s\\%s\t%d", m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                        m_Storage->FamilyName(), CurrLevel);
                if (CurrLevel > BestFinds[0].Rate)
                {
                        sprintf(BestFinds[0].String, "%s\\%s",
                                m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir, m_Storage-
>FamilyName());
                        BestFinds[0].Rate = CurrLevel;
                        bResult = TRUE;
                }
        }
}
```

```
                        else
                        {       // include to sorted list
                                if (CurrLevel > FindWorstRate)
                                {
                                        printf("\n%s\\%s\t%d", m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                                        m_Storage->FamilyName(), CurrLevel);
                                        bResult = TRUE;

                                        // find room for new element
                                        InsertIndex = -1;
                                        if ( NumBest < m_NumBest)
                                                InsertIndex = NumBest++;
                                        else
                                        {       // find worst
                                                MinRate = 100;
                                                for (i=0; i<m_NumBest; i++)
                                                {
                                                        if (BestFinds[ i].Rate <= MinRate)    // last is less valiable
                                                        {
                                                                MinRate = BestFinds[ i].Rate;
                                                                InsertIndex = i;
                                                        }
                                                }
                                                FindWorstRate = MinRate;
                                                if (CurrLevel <= FindWorstRate)
                                                        InsertIndex = -1;
                                        }
                                        if ( InsertIndex >= 0 )
                                        {
                                                BestFinds[ InsertIndex]. Rate = CurrLevel;
                                                sprintf(BestFinds[ InsertIndex].String, "%s\\%s",
                                                        m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                                        m_Storage->FamilyName());
                                        }
                                }
                        }
                }
                Count++;
                bFound = m_Storage->GetNextByFlags();
        }
        bSearchEntry = m_Storage->SetNextEntry();
}
// output report
if (m_LogFile && !m_bGuru)
{
        if (m_NumBest == 1)
                NumBest = 1;

        while(1)
        {
                MaxRate = 0;
```

```
                        MaxIndex = -1;
                        for (i=0; i < NumBest; i++)
                        {
                                if (BestFinds[ i].Next == 0 )
                                {       // was not proceeded
                                        if (BestFinds[ i].Rate > MaxRate)
                                        {
                                                MaxIndex = i;
                                                MaxRate = BestFinds[ i].Rate;
                                        }
                                }
                        }
                        if (MaxIndex >= 0)
                        {
                                bResult = TRUE;
                                fprintf(m_LogFile, "%-30s Rate=%d\n",
                                        BestFinds[ MaxIndex].String, BestFinds[ MaxIndex].Rate);
                                BestFinds[ MaxIndex].Next = 1;
                        }
                        else
                                break; // all are proceeded
                }
                if (!bResult)
                        fprintf(m_LogFile, "\nNo matches found\n");
        }
        if (!bResult)
                printf("\nNo matches found\n");

        delete BestFinds;

        return bRet;
}

/*                      if (CurrLevel > FindWorstRate)
                        {
                                printf("\n%-30s\t%d\t%d", m_Storage->FamilyName(), CurrLevel, FindWorstRate);

                                PrevIndex = -1;
                                CurrIndex = FindTopIndex;
                                while (CurrIndex >= 0)
                                {
                                        if (CurrLevel > BestFinds[ CurrIndex].Rate)
                                        {
                                                InsertIndex = CurrIndex;
                                                break;
                                        }
                                        PrevIndex = CurrIndex;
                                        CurrIndex = BestFinds[ CurrIndex ].Next;
                                }
                                // find room for new element
                                if ( NumBest < m_NumBest)
```

```
                                        InsertIndex = NumBest++;
                                else
                                {       // find worst (last in chain)
                                        InsertIndex = FindTopIndex;
                                        while ( BestFinds[ InsertIndex].Next >= 0)
                                                InsertIndex = BestFinds[ InsertIndex].Next;
                                        FindWorstRate = BestFinds[ InsertIndex].Rate;

                                        if ( FindWorstRate >= CurrLevel)
                                                InsertIndex = -1;       // don't insert
                                        else
                                                FindWorstRate = CurrLevel;
                                }
                                if ( InsertIndex >= 0 )
                                {
                                        if ( CurrIndex == FindTopIndex )
                                        {       // insert to top
                                                BestFinds[ InsertIndex ].Next = FindTopIndex;
                                                FindTopIndex = InsertIndex;
                                        }
                                        else
                                        {       // insert into chain
                                                BestFinds[ PrevIndex].Next = InsertIndex;
                                                BestFinds[ InsertIndex].Next = CurrIndex;
                                        }
                                        BestFinds[ InsertIndex]. Rate = CurrLevel;
                                        strcpy( BestFinds[ InsertIndex].String, m_Storage->FamilyName());
                                }
                        }
*/
//=========================================================
bool TMacroFamDefs::FindString( char *String )
{
        bool bRet = FALSE,
                bSearchEntry;

        int     SearchLen;

        char *SearchString = String;

        // prepare search string
        if (*SearchString == '"')
                SearchString++;

        SearchLen = strlen(SearchString);

        if (SearchString[SearchLen-1] == '"')
                SearchString[SearchLen-1] = 0;
        strupr(SearchString);

        m_Storage->SetLogFile( m_LogFile );
```

```
        // allow partial compare
        m_Storage->m_bMatches = FALSE;

        bSearchEntry = m_Storage->FirstEntry();
        while (bSearchEntry)
        {
                if (!m_Storage->OpenIndexFile() )
                        return FALSE;

                BOOL bFound = m_Storage->GetFirstByFlags( 0 );
                while ( bFound)
                {
                        // compare string
                        m_Storage->OpenDatFile();

                        if ( m_Storage->FindString( SearchString ) )
                        {
                                if (m_LogFile)
                                        fprintf(m_LogFile, "\n%s\\%s\t=> %s", m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                                m_Storage->FamilyName(), m_Storage->m_LastString);
                                printf("\n%s\\%s\t=> %s", m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                                m_Storage->FamilyName(), m_Storage->m_LastString);

                                bRet = TRUE;
                        }
                        // compare text
                        // read Lines count
                        m_Storage->ReadLineNum();

                        if ( m_Storage->FindLine( SearchString ) )
                        {
                                if (m_LogFile)
                                        fprintf(m_LogFile, "\n%s\\%s\t=> %s", m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                                m_Storage->FamilyName(), m_Storage->m_LastLine);
                                printf("\n%s\\%s\t=> %s", m_Storage->m_Entries[ m_Storage->m_CurrEntry].AliasDir,
                                                m_Storage->FamilyName(), m_Storage->m_LastLine);
                                bRet = TRUE;
                        }

                        bFound = m_Storage->GetNextByFlags();
                }
                bSearchEntry = m_Storage->NextEntry();
        }
        // output report
        if (!bRet)
                printf("\nNo matches found\n");

        return TRUE;
}
//=========================================================
bool TMacroFamDefs::Update()
```

```
{
        bool bRet = true;
        TParseInfo *pInfo;

        int     Count = 0;

        m_Storage->SetLogFile( m_LogFile );

        m_Storage->FirstEntry();
        while (m_Storage->IsEntry())
        {       // proceed one entry

                if (m_Storage->EntryCollectDir())
                {
                        m_Storage->ResetIndexFile();

                        m_Storage->FirstScanItem();
                        while (m_Storage->IsScanItem())
                        {       // proceed one item

                                printf("\n%d: %s", Count, m_Storage->ScanDirName());

                                m_Parser->Init();

                                //if (Count == 50000) //!!!
                                //{

                                m_Parser->ParseDir( m_Storage->ScanDirName() );

                                pInfo = m_Parser->GetParseInfo();

                                if (pInfo->FilesNum == 0)
                                {
                                        if (m_LogFile)
                                                fprintf(m_LogFile, "\n Empty dir: %-50s", m_Storage->ScanDirName());
                                }
                                else
                                {
                                        if (pInfo->StringsNum == 0)
                                        {
                                                if (m_LogFile)
                                                        fprintf(m_LogFile, "\n No strings: %-50s", m_Storage->ScanDirName());
                                        }
                                        else
                                        {
                                                // store item info
                                                m_Storage->PutItemHeader( pInfo->ReplFlags );

                                                if (pInfo->ReplFlags == 0)
                                                {
                                                        if (m_LogFile)
```

```
                                        fprintf(m_LogFile, "\n Unknow repl: %-50s", m_Storage->ScanDirName());
                                }
                                int     CurrIndex;
                                // store strings in alphabetic order
                                m_Storage->PutItemStringsCount( pInfo->StringsNum);
                                CurrIndex = pInfo->TopString;
                                while ( CurrIndex >= 0)
                                {
                                        m_Storage->PutItemString( pInfo->Strings[CurrIndex].String);

                                        CurrIndex = pInfo->Strings[ CurrIndex ].Next;        // next index in chain
                                }
                                m_Storage->PutItemStringsCount( pInfo->LinesNum );
                                // store strings in alphabetic order
                                CurrIndex = pInfo->TopLine;
                                while ( CurrIndex >= 0)
                                {
                                        m_Storage->PutItemString( pInfo->Lines[CurrIndex].String);

                                        CurrIndex = pInfo->Lines[ CurrIndex ].Next;  // next index in chain
                                }
                                }
                        }
                        //} //!!!
                        Count++;

                        //if (Count > 15)
                        //      break;

                        m_Storage->NextScanItem();
                }
                m_Storage->CompleteHeadersFlags(); // complete flags for last tree

                m_Storage->CloseIndexFile();
        }

        m_Storage->NextEntry();
    }
    return bRet;
}

//=============================================================
bool TMacroFamDefs::Check()
{
    bool bRet = true;
    TParseInfo *pInfo;
    char    ScanFamilyName[256];

    int     CurrIndex, ListIndex,
            StringLevel, StringCount,
            TextLevel,      TextCount,
```

```
        iSame, iSameText,
        ItemCount = 0, FileCount = 0,
        PathLen;

if (m_LogFile)
{
        fprintf(m_LogFile, "\nCheck level: replication=%d , strings=%d, text=%d\n",
                m_DetectLevelRepl, m_DetectLevelString, m_DetectLevelText);
}

m_Storage->FirstEntry();
while (m_Storage->IsEntry())
{       // proceed one entry
        PathLen = strlen( m_Storage->EntryCollectDir()) + 1;
        if (m_LogFile)
                fprintf(m_LogFile, "\nPath = %s\n", m_Storage->EntryCollectDir());

        if (!m_Storage->OpenIndexFile() )
                return FALSE;

        m_Storage->FirstScanItem();
        while (m_Storage->IsScanItem())
        {
                // proceed one item
                ItemCount++;
                printf("\nDir. %d: %s", ItemCount, m_Storage->ScanDirName());

                strcpy( ScanFamilyName, m_Storage->ScanFamilyName());
                // delete last level
                for (int i = strlen(ScanFamilyName); i>0; i--)
                {
                        if (ScanFamilyName[i] == '\\')
                        {
                                ScanFamilyName[i] = 0;
                                break;
                        }
                }
                TFileObj        fobj;
                fobj.GetFirstInside( m_Storage->ScanDirName(), "*.*");
                while (fobj.Exist())
                {
                        if (!fobj.IsDir())
                        {       // proc one file
                                FileCount++;

                                printf("\n\tFile %d: %s", FileCount, fobj.Name());

                                m_Parser->Init();
                                if (m_Parser->ParseFile( fobj.Name() ) )
                                {
                                        pInfo = m_Parser->GetParseInfo();
```

```
if (m_DetectLevelRepl == 0)
        pInfo->ReplFlags = 0;          // reset all methods

BOOL bFound = m_Storage->GetFirstByFlags( pInfo->ReplFlags );
while ( bFound)
{
        if (strncmp( ScanFamilyName, m_Storage->FamilyName(), strlen(ScanFamilyName) )
!= 0)
        {
                // not same family

                // compare string
                iSame = 0;
                CurrIndex = pInfo->TopString;
                ListIndex = 0;

                m_Storage->OpenDatFile();

                StringCount = 0;
                while ( CurrIndex >= 0)
                {
                        if (strlen(pInfo->Strings[ CurrIndex ].String) >=
(size_t)m_StringMinLen)
                        {
                                if ( m_Storage->FindString( pInfo->Strings[ CurrIndex
].String ) )
                                        iSame++;
                                StringCount++;
                        }
                        CurrIndex = pInfo->Strings[ CurrIndex ].Next;        // next
index in chain

                        ListIndex++;
                }
                StringLevel =  StringCount != 0 ? (iSame * 100)/ StringCount : 0;

                // compare text
                iSameText = 0;
                TextCount = 0;
                if ( m_DetectLevelText > 0)
                {
                        // read Lines counr
                        m_Storage->ReadLineNum();

                        CurrIndex = pInfo->TopLine;
                        while ( CurrIndex >= 0)
                        {
                                if ( m_Storage->FindLine( pInfo->Lines[ CurrIndex
].String ) )
                                        iSameText++;
                                TextCount++;
```

```
                                                        CurrIndex = pInfo->Lines[ CurrIndex ].Next;  // next
index in chain
                                                                }
                                                        }
                                                        TextLevel = TextCount != 0 ? (iSameText * 100)/ TextCount : 0;

                                                        if (StringLevel >= m_DetectLevelString &&
                                                                TextLevel >= m_DetectLevelText)
                                                        {
                                                                if (m_LogFile)
                                                                {
                                                                        fprintf(m_LogFile, "\n%-30s %-6s  %-20s  %3d/%3d ,
%3d/%3d",
                                                                                fobj.Name() + PathLen,
                                                                                TextLevel == 100 ? "Found:" : "Like :",
                                                                                m_Storage->FamilyName(),
                                                                                iSame, StringCount, iSameText, TextCount);
                                                                }
                                                        }
                                                }// not same
                                                bFound = m_Storage->GetNextByFlags();
                                        }//found
                                }// success parsing

                                }
                                fobj.GetNext();
                        }
                        m_Storage->NextScanItem();
                }
                m_Storage->CloseIndexFile();

                m_Storage->NextEntry();
        }
        return bRet;
}
//=============================================================
bool TMacroFamDefs::List( char *ListName)
{
        FILE *ListFile = fopen(ListName, "w");
        char    Tabs[32];

        m_Storage->FirstEntry();
        while (m_Storage->IsEntry())
        {       // proceed one entry
                fprintf(ListFile, "\n\nName = %s , Path = %s\n",
                        m_Storage->EntryAliasDir(), m_Storage->EntryCollectDir());

                if (!m_Storage->OpenIndexFile() )
                        return FALSE;

                BOOL bFound = m_Storage->GetFirstByFlags( 0 );
```

```
                TStoredHeader  *pHead;
                while ( bFound)
                {
                        // proceed one item
                        *Tabs = 0;
                        for (int i=0; i <= m_Storage->m_HeaderLevel; i++)
                        {
                                pHead = &(m_Storage->m_Headers[ i]);
                                if (*pHead->Name)
                                {       // was not print
                                        fprintf(ListFile, "\n%05X, %01X, %s%-30s, %04X, %05X, %01X-%06X",
                                                pHead->IndexOffset, pHead->Header.Level,
                                                Tabs, pHead->Name,
                                                pHead->Header.ReplFlags, pHead->Header.NextSibling,
                                                pHead->Header.Cluster, pHead->Header.datOffset);
                                        *pHead->Name = 0;
                                }
                                strcat( Tabs, "  ");
                        }
                        bFound = m_Storage->GetNextByFlags();
                }
                m_Storage->CloseIndexFile();

                m_Storage->NextEntry();
        }
        fclose( ListFile );
        return TRUE;
}
//=============================================================
char *TMacroFamDefs::ErrMess(int ErrCode )
{
        return m_ErrMessage;
}
//=============================================================
//=============================================================
```

```
// mfamstor.cpp - class for macrofam storage
// Author - Viatcheslav Peternev (Network Associates,Inc)
//=========================================================
#include <windows.h>

#include "MFamStor.h"


//=========================================================
// constructor
TMacroFamStorage::TMacroFamStorage()
{
        m_EntryNum = m_CurrEntry = 0;

        m_LevelNum = m_CurrLevel = 0;

        m_HeaderLevel = -1;

        m_bIsEntry = m_bIsScanItem = FALSE;

        m_DirName       = NULL;

        m_hDatFile = -1;

        m_LogFile = NULL;

        m_MaxClusterSize = MaxClusterSize;

        for (int i=0; i < MaxLevelNum; i++)
                m_Dirs[i] = NULL;

        m_XORKey = 0xD7;

        m_ErrCode = 0;
        m_ErrMessage[0] = 0;

        m_bMatches = TRUE;
}
//=========================================================
// destructor
TMacroFamStorage::~TMacroFamStorage()
{
        int i;
        for (i=0; i<m_EntryNum; i++)
        {
                if ( m_Entries[i].AliasDir)
                        delete m_Entries[i].AliasDir;
                if ( m_Entries[i].CollectDir)
                        delete m_Entries[i].CollectDir;
        }
        for (i=0; i < MaxLevelNum; i++)
```

```
        {
                if (m_Dirs[i])
                        delete m_Dirs[i];
        }
        if (m_DirName)
                delete m_DirName;

}
//=============================================================
char *TMacroFamStorage::ErrMess(int ErrCode )
{
        return m_ErrMessage;
}
//=============================================================
bool TMacroFamStorage::SetLogFile( FILE *LogFile)
{
        m_LogFile = LogFile;

        return m_LogFile != NULL;
}
//=============================================================
bool TMacroFamStorage::ResetIndexFile()
{
        if (m_CurrEntry < 0)
                return FALSE;

        sprintf(m_strFileName, "%s\\%s", m_DirName, m_Entries[m_CurrEntry].AliasDir);
        if (_access( m_strFileName, 0) != 0)
                CreateDirectory( m_strFileName, NULL );

        strcat( m_strFileName, "\\root.idx");

        m_hIndexFile = open(m_strFileName, _O_CREAT | _O_TRUNC | _O_BINARY | _O_RDWR,
                                                _S_IREAD | _S_IWRITE);

        m_CurrClusterInd = 1;
        m_CurrClusterSize = 0;

        ResetDatFile( m_CurrClusterInd );

        m_HeaderLevel = -1;

        return m_hIndexFile != -1;
}
//=============================================================
bool TMacroFamStorage::OpenIndexFile()
{
        if (m_CurrEntry < 0)
                return FALSE;

        sprintf(m_strFileName, "%s\\%s\\root.idx", m_DirName, m_Entries[m_CurrEntry].AliasDir);
```

```
        m_CurrIndexOffset = 0xFFFFFFFF;

        m_CurrDatNum = 0xFFFFFFFF;

        m_hIndexFile = open(m_strFileName, _O_BINARY | _O_RDONLY, _S_IREAD );
        if ( m_hIndexFile == -1)
        {
                m_ErrCode = ErrOpenIndex;
                sprintf( m_ErrMessage, "Error open index file %", m_strFileName);
        }

        //if (m_Max CurrIndexOffsetreturn m_hIndexFile != -1;
        //m_Max CurrIndexOffsetreturn m_hIndexFile != -1;

        return m_hIndexFile != -1;
}
//=============================================================
bool TMacroFamStorage::ResetDatFile( long Index )
{
        if (m_CurrEntry < 0)
                return FALSE;

        if (m_hDatFile >= 0)
                close(m_hDatFile);

        sprintf(m_strFileName, "%s\\%s\\%08X.dat", m_DirName,
                                m_Entries[m_CurrEntry].AliasDir, Index);

        m_hDatFile = open(m_strFileName, _O_CREAT | _O_TRUNC | _O_BINARY | _O_RDWR,
                                                _S_IREAD | _S_IWRITE);

        m_CurrClusterSize = 0;

        return m_hDatFile != -1;
}
//=============================================================
bool TMacroFamStorage::OpenDatFile()
{
        if (m_CurrDatNum != m_Headers[ m_HeaderLevel ].Header.Cluster)
        {       // doesnot already opened
                if (m_CurrEntry < 0)
                        return FALSE;

                if (m_hDatFile >= 0)
                        close(m_hDatFile);

                m_CurrDatNum = m_Headers[ m_HeaderLevel ].Header.Cluster;

                sprintf(m_strFileName, "%s\\%s\\%08X.dat", m_DirName,
```

```
                                        m_Entries[m_CurrEntry].AliasDir, m_CurrDatNum);

                m_hDatFile = open(m_strFileName, _O_BINARY | _O_RDONLY, _S_IREAD );
        }
        if ( m_hDatFile == -1)
                return FALSE;

        lseek(m_hDatFile, m_Headers[ m_HeaderLevel].Header.datOffset, SEEK_SET);

        m_ReadStringNum = 0;
        m_StringsNum = 0;
        if (read( m_hDatFile, &m_StringsNum, 2) != 2)
                return FALSE;

        return TRUE;
}
//============================================================
bool TMacroFamStorage::ReadLineNum()
{
        m_ReadLineNum = 0;
        m_LinesNum = 0;
        return (read( m_hDatFile, &m_LinesNum, 2) == 2);
}
//============================================================
bool TMacroFamStorage::CloseIndexFile()
{
        if (m_hIndexFile == -1)
                return FALSE;

        close( m_hIndexFile );

        if (m_hDatFile >= 0)
                close(m_hDatFile);

        return TRUE;
}
//============================================================
bool TMacroFamStorage::Open(char *Name)
{
        int     PosDelim, CurrOffset,
                iBeg, iEnd;
        TTextFile List;

        m_DirName = strdup(Name);

        sprintf(List.m_Line, "%s\\macrofam.map", Name);
        if (!List.OpenRead( List.m_Line))
        {
                m_ErrCode = ErrOpenBase;
                sprintf( m_ErrMessage, "Error open file %s", List.m_Line);
                return FALSE;
```

```
        }

        while (List.GetLine())
        {
                if (m_EntryNum < MaxEntryNum && strlen(List.m_Line) > 0 &&
                        *List.m_Line != ';')
                {
                        m_Entries[m_EntryNum].Type = NULL;
                        m_Entries[m_EntryNum].NextAliasDir = NULL;
                        m_Entries[m_EntryNum].CollectDir = NULL;
                        m_Entries[m_EntryNum].AliasDir = NULL;

                        // get type
                        // in first item can be spaces (Generic VBA5) !
                        GetRangeByDelims(List.m_Line, iBeg, iEnd, strlen( List.m_Line), ","); //" \t,");
                        AllocSubstring( &(m_Entries[m_EntryNum].Type), List.m_Line + iBeg, iEnd - iBeg + 1);
                        StrTrimRight(m_Entries[m_EntryNum].Type);
                        if ((PosDelim = InStr(List.m_Line, ',')) > 0)
                        {
                                CurrOffset = PosDelim + 1;
                                // get alias
                                GetRangeByDelims(List.m_Line + CurrOffset, iBeg, iEnd, strlen( List.m_Line + CurrOffset), " \t,");
                                AllocSubstring( &(m_Entries[m_EntryNum].AliasDir), List.m_Line + CurrOffset + iBeg, iEnd - iBeg + 1);

                                if ((PosDelim = InStr(List.m_Line + CurrOffset, ',')) > 0)
                                {
                                        CurrOffset += (PosDelim + 1);
                                        // get dir
                                        GetRangeByDelims(List.m_Line + CurrOffset, iBeg, iEnd, strlen( List.m_Line + CurrOffset), "
\t,");
                                        AllocSubstring( &(m_Entries[m_EntryNum].CollectDir), List.m_Line + CurrOffset + iBeg, iEnd -
iBeg + 1);

                                        if ((PosDelim = InStr(List.m_Line + CurrOffset, ',')) > 0)
                                        {
                                                CurrOffset += (PosDelim + 1);
                                                // get next alias
                                                GetRangeByDelims(List.m_Line + CurrOffset, iBeg, iEnd, strlen( List.m_Line +
CurrOffset), " \t,");
                                                AllocSubstring( &(m_Entries[m_EntryNum].NextAliasDir), List.m_Line + CurrOffset +
iBeg, iEnd - iBeg + 1);
                                        }
                                }
                        }

                        m_Entries[m_EntryNum].Checked = FALSE;
                        m_EntryNum++;
                }
        }
        List.Close();
```

```
        return TRUE;
}
//=============================================================
void TMacroFamStorage::AllocSubstring(char **Substring, char *String, int Len)
{
        *Substring = new char[Len + 1];
        memcpy( *Substring, String, Len);
        (*Substring)[Len] = 0;
}
//=============================================================
bool TMacroFamStorage::FirstEntry()
{
        if (m_EntryNum == 0)
                m_bIsEntry = FALSE;
        else
        {
                m_CurrEntry = 0;
                m_bIsEntry = TRUE;
        }
        return m_bIsEntry;
}
//=============================================================
bool TMacroFamStorage::NextEntry()
{
        if (m_EntryNum == 0 || m_CurrEntry == (m_EntryNum - 1))
                m_bIsEntry = FALSE;
        else
        {
                m_bIsEntry = TRUE;
                m_CurrEntry++;
        }
        return m_bIsEntry;
}
//=============================================================
bool TMacroFamStorage::SetEntry( char *Type)
{
        m_bIsEntry = FALSE;
        for (int i=0; i < m_EntryNum; i++)
        {
                if (strcmp(m_Entries[i].Type, Type)==0)
                {
                        m_bIsEntry = TRUE;
                        m_CurrEntry = i;
                        m_Entries[i].Checked = TRUE;
                        break;
                }
        }
        return m_bIsEntry;
}
//=============================================================
bool TMacroFamStorage::SetNextEntry()
```

```
{
        m_bIsEntry = FALSE;
        if (m_CurrEntry >=0 && m_Entries[m_CurrEntry].NextAliasDir)
        {
                for (int i=0; i < m_EntryNum; i++)
                {
                        if (i != m_CurrEntry &&
                                !m_Entries[i].Checked &&
                                strcmp(m_Entries[i].AliasDir, m_Entries[m_CurrEntry].NextAliasDir)==0)
                        {
                                m_bIsEntry = TRUE;
                                m_CurrEntry = i;
                                break;
                        }
                }
        }
        return m_bIsEntry;
}
//=============================================================
bool TMacroFamStorage::FirstScanItem()
{
        m_CurrLevel = 0;

        return ProcDir( TRUE );
}
//=============================================================
bool TMacroFamStorage::NextScanItem()
{
        return ProcDir( FALSE );
}
//=============================================================
bool TMacroFamStorage::ProcDir( bool bFirst)
{
        bool bPrev = FALSE;

        if (bFirst)
        {
                if (m_Dirs[ m_CurrLevel ])
                        delete m_Dirs[ m_CurrLevel ];
                m_Dirs[ m_CurrLevel ] = new TFileObj;

                if (m_CurrLevel == 0)
                        m_Dirs[m_CurrLevel]->GetFirstInside( m_Entries[m_CurrEntry].CollectDir, "*.*" );
                else
                {
                        m_Dirs[m_CurrLevel]->GetFirstInside( m_Dirs[m_CurrLevel - 1]->Name(), "*.*" );
                }
                bFirst = TRUE;
        }
        else
                m_Dirs[m_CurrLevel]->GetNext();
```

```
if ( m_Dirs[m_CurrLevel]->Exist() && *m_Dirs[m_CurrLevel]->Name() == '_')
        m_Dirs[m_CurrLevel]->GetNext();       // skip underscored names !

if ( m_Dirs[m_CurrLevel]->Exist())
{
        if ( m_Dirs[m_CurrLevel]->IsDir())
        {       // try next level
                if (m_CurrLevel < ( MaxLevelNum -1))
                {
                        m_CurrLevel++;
                        m_bIsScanItem = ProcDir( TRUE );
                }
                else
                        m_bIsScanItem = FALSE;
        }
        else
        {
                if (bFirst)
                {       // only files - up 1 level
                        if (m_CurrLevel > 0)
                        {
                                m_CurrLevel--;
                                m_bIsScanItem = TRUE;
                        }
                        else
                                m_bIsScanItem = FALSE;
                }
                else
                {       // mix dir & files
                        bPrev = TRUE;
                        if (m_LogFile)
                                fprintf(m_LogFile, "\nMix:%s", m_Dirs[m_CurrLevel]->Name());
                }
        }
}
else
        bPrev = TRUE;

if (bPrev)
{       // try previous level
        if (m_CurrLevel > 0)
        {
                m_CurrLevel--;
                m_bIsScanItem = ProcDir( FALSE );
        }
        else
                m_bIsScanItem = FALSE;
}
return m_bIsScanItem;
}
```

```
//=========================================================
bool    TMacroFamStorage::PutItemHeader( unsigned long Flags )
{
        TItemHeader    header;
        int iLev, iLev2;
        long    FilePos = tell( m_hIndexFile );
        unsigned long TempFlags;

        // complete previous headers
        iLev2 = 0;
        for (iLev = 0; iLev <= m_HeaderLevel; iLev++)
        {
                if ((iLev > m_CurrLevel) ||
                        (strcmp(m_Headers[iLev].Name, StrFileName( m_Dirs[iLev]->Name(), m_strFileName)) != 0) )
                {       // Upper difference
                        TempFlags = 0;
                        for (iLev2 = m_HeaderLevel; iLev2 >= iLev ; iLev2--)
                        {       // add previos flags
                                TempFlags |= m_Headers[iLev2].Header.ReplFlags;
                                if (TempFlags != m_Headers[iLev2].Header.ReplFlags)
                                {       // replace flag
                                        lseek(m_hIndexFile, m_Headers[iLev2].IndexOffset + HeaderFlagsOff, SEEK_SET);
                                        write( m_hIndexFile, &TempFlags, sizeof(TempFlags));
                                }
                        }
                        // replace sibling
                        lseek(m_hIndexFile, m_Headers[iLev].IndexOffset + 1, SEEK_SET);
                        write( m_hIndexFile, &FilePos, sizeof(FilePos));
                        iLev2 = __min(iLev, m_CurrLevel);
                        break;
                }
        }
        // write new headers
        lseek(m_hIndexFile, 0, SEEK_END);

        if (m_CurrClusterSize > m_MaxClusterSize )
        {       // create dat-file
                m_CurrClusterInd++;
                ResetDatFile( m_CurrClusterInd );
        }

        for (iLev = iLev2; iLev <= m_CurrLevel; iLev++)
        {
                memset(&header, 0, sizeof(header));

                header.Level            = iLev;
                if (iLev == m_CurrLevel)
                {
                        header.ReplFlags        = Flags;
                        header.Cluster          = m_CurrClusterInd;
                        header.datOffset        = tell(m_hDatFile);
```

```
                }
                StrFileName( m_Dirs[iLev]->Name(), m_strFileName);

                header.Len      = strlen( m_strFileName );

                // save stored headers
                strcpy(m_Headers[iLev].Name, m_strFileName);
                m_Headers[iLev].Header          = header;
                m_Headers[iLev].IndexOffset     = tell( m_hIndexFile );

                // write header
                write(m_hIndexFile, &header, sizeof(TItemHeader));
                // write name
                write(m_hIndexFile, m_strFileName, strlen( m_strFileName ));

        }
        m_HeaderLevel = m_CurrLevel;

        return TRUE;
}
//============================================================
// complete flag for previous headers
bool    TMacroFamStorage::CompleteHeadersFlags()
{
        int iLev;
        unsigned long TempFlags;

        TempFlags = 0;
        for (iLev = m_HeaderLevel; iLev >= 0 ; iLev--)
        {       // add previos flags
                TempFlags |= m_Headers[iLev].Header.ReplFlags;
                if (TempFlags != m_Headers[ iLev].Header.ReplFlags)
                {       // replace flag
                        lseek(m_hIndexFile, m_Headers[ iLev].IndexOffset + HeaderFlagsOff, SEEK_SET);
                        write( m_hIndexFile, &TempFlags, sizeof(TempFlags));
                }
        }
        return TRUE;
}
//============================================================
bool    TMacroFamStorage::PutItemStringsCount( UINT16 Count )
{
        m_CurrClusterSize += sizeof(UINT16);

        return ( write(m_hDatFile, &Count, sizeof(UINT16)) == sizeof(UINT16));

}
//============================================================
bool    TMacroFamStorage::PutItemString( char *String )
{
        UINT8 Len = (UINT8)strlen(String);
```

```
        write(m_hDatFile, &Len, sizeof(UINT8));

        if (m_XORKey)
        {
                for (int i=0; i<Len; i++)
                        String[i] ^= m_XORKey;
        }
        write(m_hDatFile, String, Len);

        m_CurrClusterSize += (Len+1);

        return TRUE;
}
//=============================================================
char    *TMacroFamStorage::ScanDirName()
{
        if (!m_bIsScanItem)
                return "";
        else
                return m_Dirs[m_CurrLevel]->Name();
}
//=============================================================
bool    TMacroFamStorage::GetFirstByFlags( unsigned long Flags )
{
        m_SearchFlags = Flags;
        m_HeaderLevel = 0;

        m_CurrIndexOffset = 0;
        lseek(m_hIndexFile, m_CurrIndexOffset, SEEK_SET);

        return GetByFlags();
}
//=============================================================
bool    TMacroFamStorage::GetNextByFlags()
{
        // find existing sibling
        for ( ; m_HeaderLevel >= 0 && m_Headers[ m_HeaderLevel ].Header.NextSibling == 0; m_HeaderLevel--) ;

        if (m_HeaderLevel < 0)
                return FALSE;

        m_CurrIndexOffset = m_Headers[ m_HeaderLevel ].Header.NextSibling;
        lseek(m_hIndexFile, m_CurrIndexOffset, SEEK_SET);

        return GetByFlags();
}
//=============================================================
bool    TMacroFamStorage::GetByFlags()
{
        bool    bFound = FALSE;
```

```
        while ( ReadIndexItem( &(m_Headers[ m_HeaderLevel ].Header),
                                m_Headers[ m_HeaderLevel ].Name, m_Headers[ m_HeaderLevel ].IndexOffset) )
        {
                if (m_Headers[ m_HeaderLevel ].Header.Level != m_HeaderLevel )
                {
                        break;
                }
                if (m_SearchFlags == 0 || m_Headers[ m_HeaderLevel ].Header.ReplFlags & m_SearchFlags)
                {       // try next level
                        bFound = TRUE;
                        m_HeaderLevel++;
                }
                else
                {       // seek to sibling
                        if (m_Headers[ m_HeaderLevel ].Header.NextSibling)
                        {
                                m_CurrIndexOffset = m_Headers[ m_HeaderLevel ].Header.NextSibling;
                                lseek(m_hIndexFile, m_CurrIndexOffset, SEEK_SET);
                        }
                        else
                                break;
                }
        }
        if (bFound)
                m_HeaderLevel--;

        return bFound;
}
//=============================================================
bool    TMacroFamStorage::ReadIndexItem( TItemHeader *pHeader, char *NameBuf, unsigned long& IndexOffset)
{
        long    ReadLen;

        // save offset
        IndexOffset = tell(m_hIndexFile);

        // read header
        ReadLen = read(m_hIndexFile, pHeader, sizeof(TItemHeader));
        if (ReadLen == sizeof(TItemHeader) )
        {       // read name
                ReadLen = read(m_hIndexFile, NameBuf, pHeader->Len);
                if (ReadLen == pHeader->Len)
                {
                        NameBuf[pHeader->Len] = 0;
                        return TRUE;
                }
        }
        return FALSE;
}
//=============================================================
bool    TMacroFamStorage::ReadString( char *Buf )
```

```
{
        if (m_ReadStringNum == m_StringsNum)
                return FALSE;

        m_ReadStringNum++;

        return ReadItem( Buf);
}
//============================================================
bool    TMacroFamStorage::ReadLine( char *Buf )
{
        if (m_ReadLineNum == m_LinesNum)
                return FALSE;

        m_ReadLineNum++;

        return ReadItem( Buf);
}
//============================================================
bool    TMacroFamStorage::ReadItem( char *Buf )
{
        long    ReadLen;
        unsigned char Len;

        // read len
        if ((ReadLen = read(m_hDatFile, &Len, 1)) == 1)
        {       // read string
                if ((ReadLen = read(m_hDatFile, Buf, Len)) == Len)
                {
                        Buf[Len] = 0;

                        if (m_XORKey)
                        {
                                for (int i=0; i<Len; i++)
                                        Buf[i] ^= m_XORKey;
                        }
                        return TRUE;
                }
        }
        return FALSE;
}
//============================================================
// Find string (counts on fact that stored and searched strings
//                              are in alphabetic order)
bool    TMacroFamStorage::FindString( char *String )
{
        bool bFound = FALSE;
        int Compare;

        if (m_ReadStringNum == 0)
                ReadString( m_LastString);
```

```
        while ( m_ReadStringNum <= m_StringsNum)
        {
                if (m_bMatches)
                {       // compare whole string
                        Compare = strcmp(String, m_LastString);
                        if (Compare == 0)
                        {       // equal
                                bFound = TRUE;
                                break;
                        }
                        else if (Compare < 0)
                        {       // less when last readed
                                break;
                        }
                        // last readed is less - try read new
                        if (!ReadString( m_LastString))
                                break;
                }
                else
                {       // partial compare
                        if (strstr(m_LastString, String))
                        {
                                bFound = TRUE;
                                break;
                        }
                        if (!ReadString( m_LastString))
                                break;
                }
        }
        return bFound;
}
//=============================================================
// Find string (counts on fact that stored and searched strings
//                          are in alphabetic order)
bool    TMacroFamStorage::FindLine( char *String )
{
        bool bFound = FALSE;
        int Compare;

        if (m_ReadLineNum == 0)
                ReadLine( m_LastLine);

        while ( m_ReadLineNum <= m_LinesNum)
        {
                if (m_bMatches)
                {       // compare whole string
                        Compare = strcmp(String, m_LastLine);
                        if (Compare == 0)
                        {       // equal
                                bFound = TRUE;
```

```
                        break;
                    }
                    else if (Compare < 0)
                    {       // less when last readed
                        break;
                    }
                    // last readed is less - try read new
                    if (!ReadLine( m_LastLine))
                        break;
                }
                else
                {       // partial compare
                    if (strstr(m_LastLine, String))
                    {
                        bFound = TRUE;
                        break;
                    }
                    if (!ReadString( m_LastLine))
                        break;
                }
            }
            return bFound;
        }
        //=============================================================
        char    *TMacroFamStorage::FamilyName()
        {
            m_strFileName[0] = 0;

            for (int i=0; i<= m_HeaderLevel; i++)
            {
                strcat(m_strFileName, m_Headers[ i ].Name);
                if (i<m_HeaderLevel)
                    strcat(m_strFileName, "\\");
            }
            return m_strFileName;
        }
        //=============================================================
        char    *TMacroFamStorage::ScanFamilyName()
        {
            m_strFileName[0] = 0;

            for (int i=0; i<= m_CurrLevel; i++)
            {
                StrFileName( m_Dirs[ i ]->Name(),  m_strFileName + strlen(m_strFileName));
                if (i < m_CurrLevel)
                    strcat(m_strFileName, "\\");
            }
            return m_strFileName;
        }
```

```
// mfampars.cpp - parser for macro sources
// Author - Viatcheslav Peternev (Network Associates,Inc)
//=========================================================
#include "MFamPars.h"

//=========================================================
// constructor
TMacroFamParser::TMacroFamParser()
{
        m_pVBAMod = new  VBAModules;

        // init strings storage
        m_MaxStringsBufLen = MaxStringLen * MaxStringNum;
        m_StringBuf     = new char[MaxStringLen];
        m_StringsBuf= new char[m_MaxStringsBufLen];

        // init lines storage
        m_MaxLinesBufLen = MaxLineLen * MaxLineNum;
        m_LineBuf       = new char[MaxLineLen];
        m_LinesBuf      = new char[m_MaxLinesBufLen];

        ResetStrings();
}
//=========================================================
// destructor
TMacroFamParser::~TMacroFamParser()
{
        delete m_LineBuf;
        delete m_StringBuf;
        delete m_LinesBuf;
        delete m_StringsBuf;
        delete m_pVBAMod;

}
//=========================================================
void TMacroFamParser::ResetStrings()
{
        //for (int i= 0; i < m_StringsNum; i++)
        //      delete m_Strings[i].String;

        m_StringsNum = 0;
        m_TopString     = -1;
        m_StringsBufLen = 0;

        m_LinesNum      = 0;
        m_TopLine       = -1;
        m_LinesBufLen = 0;
}
//=========================================================
char    *TMacroFamParser::ErrMess(int ErrCode )
{
```

```
        return "";
}
//=========================================================
bool    TMacroFamParser::ParseDir( char *DirName)
{
        TFileObj        fobj;

        m_FilesNum      = 0;

        fobj.GetFirstInside( DirName, "*.*");
        while (fobj.Exist())
        {
                if (!fobj.IsDir())
                        ParseFile( fobj.Name() );

                fobj.GetNext();
        }
        return true;
}
//=========================================================
void    TMacroFamParser::Init()
{
        m_ReplFlags = 0;
        ResetStrings();
        for (int i= 0; i < MaxTypesNum; i++)
                m_FilesOfType[i] = 0;

}
//=========================================================
bool    TMacroFamParser::ParseFile( char *FileName)
{
        bool bRet = TRUE;

        strupr(FileName);
        if (strstr( FileName, ".BAS") || strstr( FileName, ".M") || strstr( FileName, "POINTER"))
                return FALSE;

        if (!m_pVBAMod->Open( FileName ))
                return FALSE;

        m_pVBAMod->LoadAllModules(10000, 200000);

        if (m_pVBAMod->GetModCount() > 0)
        {
                m_FilesNum++;

                if (strstr(FileName, ".DOT"))
                {
                        if (strncmp(FileName, "NORM", 4) == 0)
                                m_CurrType = 1;
                        else
```

```
                        m_CurrType = 2;
                }
                else
                        m_CurrType = 0;

                m_FilesOfType[ m_CurrType ]++;

                TVBAModInfo *pInfo = m_pVBAMod->GetFirstInfo();
                while (pInfo)
                {
                        if (pInfo->SourceLen)
                        {       // output source
                                int iBufPos = 0;
                                while (iBufPos < pInfo->SourceLen)
                                {
                                        // get line
                                        m_LineLen = 0;
                                        while (iBufPos < pInfo->SourceLen && pInfo->Source[iBufPos] != 0x0D)
                                        {
                                                if (m_LineLen < MaxLineLen)
                                                {
                                                        if (m_LineLen > 0 || (pInfo->Source[iBufPos] != ' ' &&
                                                                                pInfo->Source[iBufPos] != '\t') )
                                                                m_LineBuf[m_LineLen++] = pInfo->Source[iBufPos];
                                                }
                                                iBufPos++;
                                        }
                                        if (m_LineLen == MaxLineLen)
                                                m_LineLen--;

                                        m_LineBuf[m_LineLen] = 0;

                                        // proceed line
                                        ProcLine();

                                        iBufPos++;
                                        if (pInfo->Source[iBufPos] == 0x0A)
                                                iBufPos++;
                                }
                        }
                        pInfo = m_pVBAMod->GetNextInfo();
                }
        }
        else
                bRet = FALSE;

        m_pVBAMod->Close();

        return bRet;
}
//=============================================================
```

```
void    TMacroFamParser::ProcLine()
{
        if (m_LineLen == 0)
                return;

        if (m_LineBuf[0] == '\'')
                return;         // skip comments

        if (strncmp(m_LineBuf, "REM ", 4) == 0)
                return;         // skip comments

        strupr(m_LineBuf);

        if (strncmp(m_LineBuf, "ATTRIBUTE", 9) == 0)
                return;         // skip attribute

        if (strncmp(m_LineBuf, "END SUB", 7) == 0)
                return;         // skip end statment

        if (strstr(m_LineBuf, "\"E "))
                return;         // skip edit strings for debbugger

        if (strstr(m_LineBuf, "PUT #"))
                return;

        // check for replication words
        char *pDot;
        if (pDot = strstr(m_LineBuf, "."))
        {
                if (strstr( pDot, ".ORGANIZER"))
                        m_ReplFlags |= ReplOrganizer;
                else if (strstr(pDot, ".MACROCOPY"))
                        m_ReplFlags |= ReplMacroCopy;
                else if (strstr(pDot, ".IMPORT"))
                        m_ReplFlags |= ReplImport;
                else if (strstr(pDot, ".REPLACELINE"))
                        m_ReplFlags |= ReplReplaceLine;
                else if (strstr(pDot, ".INSERTLINES"))
                        m_ReplFlags |= ReplInsertLines;
                else if (strstr(pDot, ".ADDFROMSTRING"))
                        m_ReplFlags |= ReplAddFromString;
                else if (strstr(pDot, ".ADDFROMFILE"))
                        m_ReplFlags |= ReplAddFromFile;
        }
        // check for string constants
        char *pQuote = m_LineBuf;

        while(pQuote = strchr(pQuote, '"'))
        {
                // get line
                m_StringLen = 0;
```

```
                pQuote++;
                while ( *pQuote && *pQuote != '"')
                {
                        if (m_StringLen < MaxStringLen)
                                m_StringBuf[m_StringLen++] = *pQuote;
                        pQuote++;
                }
                if ( *pQuote )
                        pQuote++;

                if (m_StringLen == MaxStringLen)
                        m_StringLen--;

                m_StringBuf[m_StringLen] = 0;

                SaveString();
        }
        SaveLine();
}
//=========================================================
char *TMacroFamParser::Type()
{
        return m_pVBAMod->GetTypeName();
}
//=========================================================
void    TMacroFamParser::SaveLine()
{
        if (m_LinesNum >= MaxLineNum)
                return;

        // find place in chain
        int             CurrIndex = m_TopLine,
                        PrevIndex = -1,
                        Len = strlen(m_LineBuf);
        BOOL    bInsert = TRUE, bDelete = FALSE;

        if ( m_FilesOfType[ m_CurrType ] == 1)
        {       // add all line for first file of current type
                while ( CurrIndex >= 0)
                {
                        int iOrd = strcmp( m_LineBuf, m_Lines[ CurrIndex ].String);
                        if (iOrd == 0 )
                        {
                                bInsert = FALSE;
                                break;  // same string
                        }
                        else if (iOrd < 0)
                        {       // insert to chain
                                break;
                        }
                        PrevIndex = CurrIndex;
```

```
                        CurrIndex = m_Lines[ CurrIndex ].Next;        // next index in chain
                }
                if (bInsert)
                {
                        if ( (m_LinesBufLen + Len) < m_MaxLinesBufLen)
                        {
                                m_Lines[ m_LinesNum ].String = m_LinesBuf + m_LinesBufLen;
                                strcpy(m_LinesBuf + m_LinesBufLen, m_LineBuf);
                                m_LinesBufLen += (Len + 1);

                                m_Lines[ m_LinesNum ].Next = CurrIndex;
                                m_Lines[ m_LinesNum ].Type = m_CurrType;
                                m_Lines[ m_LinesNum ].Use = 1;

                                if (PrevIndex >= 0)
                                        m_Lines[ PrevIndex].Next = m_LinesNum;

                                if ( CurrIndex == m_TopLine )
                                        m_TopLine = m_LinesNum;

                                m_LinesNum++;
                        }
                }
        }
        else
        {       // delete all absent lines for next files of current type
                while ( CurrIndex >= 0)
                {
                        if ( m_CurrType == m_Lines[ CurrIndex ].Type)
                        {
                                int iOrd = strcmp( m_LineBuf, m_Lines[ CurrIndex ].String);
                                if (iOrd == 0 )
                                {
                                        if ( m_Lines[ CurrIndex ].Use < (m_FilesOfType[ m_CurrType ] - 1))
                                                m_Lines[ CurrIndex ].Use = 0;
                                        else
                                                m_Lines[ CurrIndex ].Use = m_FilesOfType[ m_CurrType ];
                                        break;  // save the same stored string
                                }
                                else if (iOrd < 0)
                                        break;  // no farther
                        }
                        // try nex stored line
                        PrevIndex = CurrIndex;
                        CurrIndex = m_Lines[ CurrIndex ].Next;        // next index in chain
                }
        }
}
//=========================================================
void    TMacroFamParser::SaveString()
{
```

```
        if (m_StringsNum >= MaxStringNum)
                return;

        // find place in chain
        int             CurrIndex = m_TopString,
                        PrevIndex = -1,
                        Len = strlen(m_StringBuf);
        BOOL    bInsert = TRUE;

        while ( CurrIndex >= 0)
        {
                int iOrd = strcmp( m_StringBuf, m_Strings[ CurrIndex ].String);
                if (iOrd == 0)
                {
                        bInsert = FALSE;
                        break;  // same string
                }
                else if (iOrd < 0)
                {       // insert to chain
                        break;
                }
                PrevIndex = CurrIndex;
                CurrIndex = m_Strings[ CurrIndex ].Next;      // next index in chain
        }
        if (bInsert)
        {
                if ( (m_StringsBufLen + Len) < m_MaxStringsBufLen)
                {
                        m_Strings[ m_StringsNum ].String = m_StringsBuf + m_StringsBufLen;

                        strcpy(m_StringsBuf + m_StringsBufLen, m_StringBuf);
                        m_StringsBufLen += (Len + 1);

                        m_Strings[ m_StringsNum ].Next = CurrIndex;
                        if (PrevIndex >= 0)
                                m_Strings[ PrevIndex].Next = m_StringsNum;

                        if ( CurrIndex == m_TopString )
                                m_TopString = m_StringsNum;

                        m_StringsNum++;
                }
        }
}
//=============================================================
TParseInfo *TMacroFamParser::GetParseInfo()
{
        m_ParseInfo.FilesNum   = m_FilesNum;
        m_ParseInfo.ReplFlags  = m_ReplFlags;

        m_ParseInfo.StringsNum = m_StringsNum;
```

```
        m_ParseInfo.Strings           = m_Strings;
        m_ParseInfo.TopString  = m_TopString;
        // count of lines which present of all files
        m_ParseInfo.Lines             = m_Lines;
        m_ParseInfo.TopLine           = m_TopLine;
        m_ParseInfo.LinesNum = 0;

        int           CurrIndex = m_TopLine,
                      PrevIndex = -1;
        bool    bDelete;
        while ( CurrIndex >= 0)
        {
              bDelete = FALSE;

              if ( m_Lines[ CurrIndex ].Use == m_FilesOfType[ m_Lines[ CurrIndex ].Type ])
              {
                    if (PrevIndex >= 0 && strcmp( m_Lines[ CurrIndex ].String, m_Lines[ PrevIndex ].String ) == 0)
                          bDelete = TRUE;
                    else
                          m_ParseInfo.LinesNum++;
              }
              else
                    bDelete = TRUE;

              if ( bDelete)
              {     // delete stored line from chain
                    if (CurrIndex == m_ParseInfo.TopLine)
                    {     // top record
                          m_ParseInfo.TopLine = m_Lines[ CurrIndex ].Next;
                          CurrIndex = m_Lines[ CurrIndex ].Next;
                    }
                    else
                    {
                          m_Lines[ PrevIndex ].Next = m_Lines[ CurrIndex ].Next;
                          CurrIndex = m_Lines[ PrevIndex ].Next;
                    }
              }
              else
              {
                    // try nex stored line
                    PrevIndex = CurrIndex;
                    CurrIndex = m_Lines[ CurrIndex ].Next;
              }
        }
        return &m_ParseInfo;
}
//=============================================================
//=============================================================
```

```
//#ifndef _COMMON_H_

#define _COMMON_H_

#define TRUE 1
#define FALSE 0


typedef char CHAR;
typedef char * LPSTR;

typedef unsigned char BYTE;
typedef BYTE * LPBYTE;

typedef unsigned short WORD;
typedef WORD * LPWORD;

//typedef unsigned int DWORD;
//typedef DWORD * LPDWORD;

typedef unsigned int  UINT;

//#endif
```

```
//-----------------------------------------------------------
//  File:    GLOBAL.H
//
//  Description:
//
//          Global type definition for all platforms
//
//-----------------------------------------------------------

#ifndef _GLOBAL_H_
#define _GLOBAL_H_

// platform definitions

#if __cplusplus
  #define _CPP_ 1
#endif

#ifdef _MSC_VER
  #define _MSC_ _MSC_VER

  #ifdef _WIN32
    #define _WIN_ 32
  #else
    #define _DOS_ 1
  #endif
#elif defined(__BORLANDC__)
  #define _BC_ __BORLANDC__

  #ifdef _Windows
    #ifdef __WIN32__
      #define _WIN_ 32
    #else
      #define _WIN_ 16
    #endif
  #else
    #define _DOS_ 1
  #endif

#else
  #error Unknown platform
#endif

// constants checking

#if defined(_MSC_) && defined(_BC_) || (defined(_DOS_) && defined(_WIN_))
  #error Ambigous platform
#endif

// alias types
```

```
#ifndef _GLOBAL_DONT_TYPEDEF
  typedef unsigned char UCHAR;
  typedef signed char SCHAR;
  typedef char CHAR;
  typedef unsigned short USHORT;
  typedef signed short SHORT;
  typedef unsigned int UINT;
  typedef signed int INT;
  typedef unsigned long ULONG;
  typedef signed long LONG;
  // bool is equal to int
  typedef INT BOOL;

  // fixed-sized types
  typedef signed char INT8;
  typedef signed short INT16;
  typedef unsigned char UINT8;
  typedef unsigned short UINT16;

  #if !_MSC_
    typedef signed long INT32;
    typedef unsigned long UINT32;

    #if _BC_ < 0x500
      typedef BOOL bool;
    #endif
  #else
    #include <basetsd.h>

    #if !_CPP_
      typedef BOOL bool;
    #endif
  #endif
#endif

// some standard constants (if they aren't defined yet)

#ifndef TRUE
  #define TRUE 1
#endif

#ifndef FALSE
  #define FALSE 0
#endif

#ifndef NULL
  #define NULL 0UL
#endif

// C-compliant func. definitions for CPP progs
```

```
#if _CPP_
  #define _C_DECLARE                    extern "C"
  #define _C_DECL_BEGIN                 _C_DECLARE {
  #define _C_DECL_END                   }
#else
  #define _C_DECLARE
  #define _C_DECL_BEGIN
  #define _C_DECL_END
#endif

#endif
```

```
// mfamdefs.h - header file for mfamdefs.cpp
// Author - Viatcheslav Peternev (Network Associates,Inc)
//=======================================================
#ifndef _T_MFAMDEFS_
#define _T_MFAMDEFS_

#include <stdio.h>

#include "MFamPars.h"
#include "MFamStor.h"

class TMacroFamDefs{
public:
        // methods
        TMacroFamDefs();                // constructor
        ~TMacroFamDefs();               // destructor

        bool OpenStorage( char *Name );
        bool SetLogFile( char *Name );
        void SetParams( FILE *inifile );

        bool FindFamily( char *FileName );
        bool FindString( char *String );
        bool Update();
        bool Check();
        bool List( char *ListName);

        char    *ErrMess(int ErrCode );
        int             ErrCode() { return m_ErrCode; }

        // ini variables
        int     m_DetectLevelRepl,
                m_DetectLevelString,
                m_DetectLevelText,
                m_NumBest,
                m_StringMinLen;
        bool m_bGuru;

private:
        // variables
        TMacroFamParser                 *m_Parser;
        TMacroFamStorage        *m_Storage;
        FILE                            *m_IniFile,
                                                *m_LogFile;
        unsigned char           m_FoundMap[256];
        int             m_ErrCode;
        char    m_ErrMessage[255];

};

#endif
```

```
// mfamstore.h - header file for mfamstore.cpp
// Author - Viatcheslav Peternev (Network Associates,Inc)
//=========================================================
#ifndef _T_MFAMSTOR_
#define _T_MFAMSTOR_

#include "t_fobj.h"
#include "t_str.h"

//#include "MFamPars.h"

typedef struct tagEntryInfo{
        char *Type;
        char *AliasDir;
        char *CollectDir;
        char *NextAliasDir;
        BOOL    Checked;
} TEntryInfo;

typedef struct tagItemHeader{
        unsigned char Level;
        unsigned long NextSibling;
        unsigned long ReplFlags;
        unsigned char Cluster;
        unsigned long DatOffset;
        unsigned char Len;
} TItemHeader;

typedef struct tagStoredHeader{
        TItemHeader    Header;
        unsigned long IndexOffset;
        char    Name[255];
} TStoredHeader;

class TMacroFamStorage{
public:
        enum
        {       ErrOpenBase = -1,
                ErrOpenIndex = -2
        };

        enum
        {       MaxEntryNum = 64,
                MaxLevelNum = 7,
                MaxClusterSize = 0x80000,
                HeaderFlagsOff = 5
        };

        // methods
        TMacroFamStorage();           // constructor
        ~TMacroFamStorage();  // destructor
```

```
bool    Open(char *Name);

char    *FamilyName();

bool    SetLogFile( FILE *LogFile);

// Index file func.
bool    ResetIndexFile();
bool    OpenIndexFile();
bool    CloseIndexFile();
bool    ReadIndexItem( TItemHeader *pHeader, char *NameBuf, unsigned long& IndexOffset);
bool    GetFirstByFlags( unsigned long Flags );
bool    GetNextByFlags();
bool    GetByFlags();
bool    PutItemHeader( unsigned long Flags );
bool    CompleteHeadersFlags();

// Dat.file func.
bool    ResetDatFile( long Offset );
bool    OpenDatFile();
bool    ReadString( char *Buf );
bool    ReadLine( char *Buf );
bool    ReadItem( char *Buf );
bool    ReadLineNum();
bool    FindString( char *String);
bool    FindLine( char *String);
bool    PutItemStringsCount( UINT16 Count );
bool    PutItemString( char *String );

// Entry functions
bool    FirstEntry();
bool    NextEntry();
bool    SetEntry( char *Type);
bool    SetNextEntry();
bool    IsEntry() { return m_bIsEntry; }
char    *EntryAliasDir() { return m_Entries[m_CurrEntry].AliasDir; }
char    *EntryCollectDir() { return m_Entries[m_CurrEntry].CollectDir; }
void    AllocSubstring(char **Substring, char *String, int Len);

// Scanning
bool    FirstScanItem();
bool    NextScanItem();
bool    IsScanItem() { return m_bIsScanItem; }
char    *ScanDirName();
bool    ProcDir( bool bFirst);
char    *ScanFamilyName();

char    *ErrMess(int ErrCode );
int             ErrCode() { return m_ErrCode; }
```

```
        // variables
        TStoredHeader  m_Headers[ MaxLevelNum ];
        int                          m_HeaderLevel;

        TEntryInfo m_Entries[ MaxEntryNum ];
        int            m_EntryNum, m_CurrEntry;

        BOOL    m_bMatches;

        char    m_strFileName[ 256 ],
                m_LastString[ 256 ],
                m_LastLine[ 256 ];

private:
        // variables
        char    *m_DirName;

        bool    m_bIsEntry,
                m_bIsScanItem;

        TFileObj        *m_Dirs[ MaxLevelNum ];

        int             m_LevelNum,
                        m_CurrLevel,
                        m_StringsNum,  m_ReadStringNum,
                        m_LinesNum,             m_ReadLineNum,
                        m_CurrClusterInd,       m_CurrClusterSize,      m_MaxClusterSize;

        int             m_hIndexFile,
                        m_hDatFile;

        unsigned char m_XORKey;

        int             m_ErrCode;

        unsigned long  m_SearchFlags,
                                        m_CurrIndexOffset,
                                        m_CurrDatNum;
        FILE                    *m_LogFile;

        char    m_ErrMessage[255];
};

#endif
```

```
// mfampars.h - header file for mfampars.cpp
// Author - Viatcheslav Peternev (Network Associates,Inc)
//=========================================================
#ifndef _T_MFAMPARS_
#define _T_MFAMPARS_

#include "vbamod.h"

#include "t_fobj.h"
#include "t_str.h"


typedef struct tagStrings{
        char                    *String;
        unsigned char  Type;
        unsigned char  Use;
        int                             Next;
} TStrings;

typedef struct tagParseInfo{
        int                     FilesNum;
        TStrings        *Strings;
        TStrings        *Lines;
        int                     StringsNum;
        int                     TopString;              // index of first string in chain
        unsigned long ReplFlags;
        int                     LinesNum;
        int                     TopLine;
} TParseInfo;

class TMacroFamParser{
public:
        enum    // constants
        {       MaxLineLen=256,
                MaxStringLen=256,
                MaxStringNum=256,
                MaxLineNum=2048,
                MaxTypesNum=3
        };
        enum    // replication flags
        {
                ReplMacroCopy  =0x00000001,
                ReplReplaceLine         =0x00000002,
                ReplInsertLines         =0x00000004,
                ReplAddFromString=0x00000008,
                ReplAddFromFile         =0x00000010,
                ReplOrganizer  =0x00000020,
                ReplImport              =0x00000040
        };
        // methods
        TMacroFamParser();              // constructor
```

```
        ~TMacroFamParser();          // destructor

        void    Init();

        char    *Type();
        char    *ErrMess(int ErrCode );
        int             ErrCode() { return m_ErrCode; }
        bool    ParseDir( char *DirName );
        bool    ParseFile( char *FileName );
        TParseInfo *GetParseInfo();

private:

        void    ProcLine();
        void    SaveLine();
        void    SaveString();

        void    ResetStrings();

        // variables
        TParseInfo      m_ParseInfo;
        VBAModules      *m_pVBAMod;

        char    *m_LineBuf,             *m_StringBuf,
                        *m_LinesBuf,    *m_StringsBuf   ;

        unsigned char  m_CurrType;
        int            m_FilesOfType[ MaxTypesNum ];

        TStrings       m_Strings[MaxStringNum];

        TStrings       m_Lines[MaxLineNum];

        int            m_LineLen,              m_StringLen,
                       m_StringsNum,   m_LinesNum,
                       m_LinesBufLen, m_StringsBufLen,
                       m_MaxLinesBufLen,       m_MaxStringsBufLen,
                       m_TopString,    m_TopLine,
                       m_FilesNum,
                       m_ErrCode;

        unsigned long m_ReplFlags;
};

#endif
```